LABORATORY FOR COMPUTER SCIENCE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MIT/LCS/TR-469

LOCALITY IN PARALLEL COMPUTATION

Bruce MacDowell Maggs

September 1989

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TR 469	5 MONITORING ORGANIZATION REPORT NUMBER(S) N00014-87-K-825
6a NAME OF PERFORMING ORGANIZATION MIT Lab for Computer Science 6b OFFICE SYMBOL (If applicable)	7a NAME OF MONITORING ORGANIZATION Office of Naval Research/Dept. of Navy
6c. ADDRESS (City, State, and ZIP Code) 545 Technology Square Cambridge, MA 02139	7b. ADDRESS (City, State, and ZIP Code) Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD 8b OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd. Arlington, VA 22217	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO
11 TITLE (Include Security Classification) Locality in Parallel Computation	
12 PERSONAL AUTHOR(S) Maggs, Bruce MacDowell	
13a. TYPE OF REPORT Technical 13b. TIME COVERED FROM ____ TO ____	14. DATE OF REPORT (Year, Month, Day) September 1989 15 PAGE COUNT 158
16. SUPPLEMENTARY NOTATION	
17 COSATI CODES FIELD / GROUP / SUB-GROUP	18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) parallel computation, fixed connection networks, packet routing algorithms, area universal networks, fat-trees; distributed random-access machines, graph algorithms, net. emu.

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This thesis explores strategies for exploiting *locality* in three major areas of parallel computation: packet routing, graph algorithms, and network emulations. Each of these areas is covered by a separate chapter.

Chapter 1 describes a network-independent approach to the packet-routing problem. Our strategy is to partition the problem into two stages: a path-selection stage and a scheduling stage. In the first stage we find paths for the packets with small congestion, c, and dilation, d. Once the paths are fixed, both are lower bounds on the time required to deliver the packets. In the second stage we find a schedule for the movement of each packet along its path so that no two packets traverse the same edge at the same time, and so that the total time and maximum queue size required to route all of the packets to their destinations are minimized.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Judy Little	22b TELEPHONE (Include Area Code) (617) 253-5894 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

U.S. Government Printing Office 1985-607-047

Unclassified

19 continued

Although path-selection strategies vary from network to network, we show that there is an efficient on-line scheduling algorithm for the entire class of layered networks. When applied to an N-packet problem, the algorithm produces a schedule of length $O(c + d + \log N)$, with high probability.

The algorithm has many applications to routing and sorting. Among them are the first on-line algorithms for routing N-packets on an N-node shuffle-exchange graph in $O(\log N)$ steps using constant-size queues and for routing kN packets on an N-node k-dimensional array with maximum side length M in $O(kM)$ steps using constant-size queues. The scheduling algorithm can also be used as a subroutine in sorting algorithms. It yields the first asymptotically optimal algorithms for sorting on butterfly, shuffle-exchange, and multidimensional array networks using constant-size queues.

The algorithm can also be applied to the construction of area-universal networks: N-node networks with VLSI-layout area $O(N)$ that can simulate all other networks with area $O(N)$ with only $O(\log N)$ slowdown.

In Chapter 1 we also prove the existence of a schedule of length $O(c + d)$ for any set of packets whose paths have congestion c and dilation d (in any network) that uses constant-size queues. Unfortunately, no efficient algorithm for constructing the schedule is known.

Chapter 2 introduces a model for parallel computation, called the *distributed random-access machine* (DRAM), in which the communication requirements of parallel algorithms can be evaluated. A DRAM is an abstraction of a parallel computer in which memory accesses are implemented by routing messages through a communication network. It explicitly models the congestion of messages across cuts of the network.

We introduce the notion of a *conservative* algorithm as one whose communication requirements at each step can be bounded by the congestion of pointers of the input data structure across cuts of a DRAM. A conservative algorithm is guaranteed not to generate undo congestion in any underlying network. Chapter 2 presents conservative algorithms for a variety of graph problems. Problems such as computing treewalk numberings, finding the separator of a tree, and evaluating all subexpressions in an expression tree can be solved in $O(\log N)$ steps for N-node trees by conservative algorithms for an exclusive-read exclusive-write DRAM. More complex problems such as finding a minimum-cost spanning forest, computing biconnected components and constructing an Eulerian cycle require $O(\log^2 N)$ steps, for graphs of size N. For concurrent-read concurrent-write DRAM's, all of these problems can be solved by $O(\log N)$ step conservative algorithms.

Chapter 3 examines the problem of how efficiently a host network can emulate a guest network. The goal is to emulate T_G steps of an N_G-node guest network on an N_H node host network. We call an emulation *work-preserving* if the time required by the host, T_H is $O(T_G N_G / N_H)$ because then both the guest and host networks perform the same amount of total work (processor–time product), $\Theta(T_G N_G)$, to within a constant factor. A work-preserving emulation is efficient because it achieves optimal speedup over a sequential emulation of the guest. We say that an emulation is *real-time* if $T_H = O(T_G)$, because then the host emulates the guest with constant delay.

Although many isolated emulation results have been proved for specific networks in the past, and measures such as dilation and congestion were known to be important, the field has lacked a model within which general results and meaningful lower bounds could be proved. We attempt to provide such a model, along with techniques for proving lower bounds based on comparing the locality the networks.

Some of the more interesting and diverse results in Chapter 3 include a proof that a linear array can emulate a (much larger) butterfly in a work-preserving fashion, but that a butterfly cannot emulate an expander (of any size) in a work-preserving fashion; a proof that a mesh can be emulated in real time in a work-preserving fashion on a butterfly, even though any $O(1)$-to-1 embedding of the mesh has dilation $\Omega(\log N)$; and a proof that an N-node butterfly can emulate an $N \log N$-node shuffle-exchange graph in a work-preserving fashion, and vice-versa.

Chapter 4 presents an algorithm for finding a minimum-cost spanning tree of an N-node graph on an $N \times N$ mesh-connected computer. The algorithm has the same $O(N)$ running time as the previous algorithms, but it is much simpler.

Locality in Parallel Computation

by

Bruce MacDowell Maggs

S.B., Computer Science and Engineering
Massachusetts Institute of Technology
(1985)
S.M., Electrical Engineering and Computer Science
Massachusetts Institute of Technology
(1986)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1989

Signature of Author_____
Department of Electrical Engineering and Computer Science
September 1, 1989

Certified by_____
Charles E. Leiserson
Associate Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by_____
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Locality in Parallel Computation

by

Bruce MacDowell Maggs

Submitted to the Department of Electrical Engineering and Computer Science
on September 1, 1989, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

This thesis explores strategies for exploiting *locality* in three major areas of parallel computation: packet routing, graph algorithms, and network emulations. Each of these areas is covered by a separate chapter.

Chapter 1 describes a network-independent approach to the packet-routing problem. Our strategy is to partition the problem into two stages: a path-selection stage and a scheduling stage. In the first stage we find paths for the packets with small congestion, c, and dilation, d. Once the paths are fixed, both are lower bounds on the time required to deliver the packets. In the second stage we find a schedule for the movement of each packet along its path so that no two packets traverse the same edge at the same time, and so that the total time and maximum queue size required to route all of the packets to their destinations are minimized.

Although path-selection strategies vary from network to network, we show that there is an efficient on-line scheduling algorithm for the entire class of layered networks. When applied to an N-packet problem, the algorithm produces a schedule of length $O(c + d + \log N)$, with high probability.

The algorithm has many applications to routing and sorting. Among them are the first on-line algorithms for routing N-packets on an N-node shuffle-exchange graph in $O(\log N)$ steps using constant-size queues and for routing kN packets on an N-node k-dimensional array with maximum side length M in $O(kM)$ steps using constant-size queues. The scheduling algorithm can also be used as a subroutine in sorting algorithms. It yields the first asymptotically optimal algorithms for sorting on butterfly, shuffle-exchange, and multidimensional array networks using constant-size queues.

The algorithm can also be applied to the construction of area-universal networks: N-node networks with VLSI-layout area $O(N)$ that can simulate all other networks with area $O(N)$ with only $O(\log N)$ slowdown.

In Chapter 1 we also prove the existence of a schedule of length $O(c + d)$ for any set of packets whose paths have congestion c and dilation d (in any network) that uses constant-size queues. Unfortunately, no efficient algorithm for constructing the schedule is known.

Chapter 2 introduces a model for parallel computation, called the *distributed random-access machine* (DRAM), in which the communication requirements of parallel algorithms can be evaluated. A DRAM is an abstraction of a parallel computer in which memory accesses are implemented by routing messages through a communication network. It explicitly models the congestion of messages across cuts of the network.

We introduce the notion of a *conservative* algorithm as one whose communication requirements at each step can be bounded by the congestion of pointers of the input data structure across cuts of a DRAM. A conservative algorithm is guaranteed not to generate undo congestion in any underlying network. Chapter 2 presents conservative algorithms for a variety of graph problems. Problems such as computing treewalk numberings, finding the separator of a tree, and evaluating all subexpressions in an expression tree can be solved in $O(\log N)$ steps for N-node trees by conservative algorithms for an exclusive-read exclusive-write DRAM. More complex problems such as finding a minimum-cost spanning forest, computing biconnected components and constructing an Eulerian cycle require $O(\log^2 N)$ steps, for graphs of size N. For concurrent-read concurrent-write DRAM's, all of these problems can be solved by $O(\log N)$ step conservative algorithms.

Chapter 3 examines the problem of how efficiently a host network can emulate a guest network. The goal is to emulate T_G steps of an N_G-node guest network on an N_H node host network. We call an emulation *work-preserving* if the time required by the host, T_H is $O(T_G N_G/N_H)$ because then both the guest and host networks perform the same amount of total work (processor-time product), $\Theta(T_G N_G)$, to within a constant factor. A work-preserving emulation is efficient because it achieves optimal speedup over a sequential emulation of the guest. We say that an emulation is *real-time* if $T_H = O(T_G)$, because then the host emulates the guest with constant delay.

Although many isolated emulation results have been proved for specific networks in the past, and measures such as dilation and congestion were known to be important, the field has lacked a model within which general results and meaningful lower bounds could be proved. We attempt to provide such a model, along with techniques for proving lower bounds based on comparing the locality the networks.

Some of the more interesting and diverse results in Chapter 3 include a proof that a linear array can emulate a (much larger) butterfly in a work-preserving fashion, but that a butterfly cannot emulate an expander (of any size) in a work-preserving fashion; a proof that a mesh can be emulated in real time in a work-preserving fashion on a butterfly, even though any $O(1)$-to-1 embedding of the mesh has dilation $\Omega(\log N)$; and a proof that an N-node butterfly can emulate an $N \log N$-node shuffle-exchange graph in a work-preserving fashion, and vice-versa.

Chapter 4 presents an algorithm for finding a minimum-cost spanning tree of an N-node graph on an $N \times N$ mesh-connected computer. The algorithm has the same $O(N)$ running time as the previous algorithms, but it is much simpler.

Keywords: parallel computation, fixed-connection networks, packet routing algorithms, area-universal networks, fat-trees, distributed random-access machines, graph algorithms, network emulations.

Thesis Supervisor: Charles E. Leiserson
Title: Associate Professor of Computer Science and Engineering

Contents

Acknowledgments

I am indebted to Charles Leiserson for teaching me the meaning of scholarship. His advice has not always been easy to take, and I have often momentarily wished that his standards were not so high. Yet, looking back over the past four years, I see that whenever Charles demanded extra effort, it was well spent. Much of the research in this thesis was done in collaboration with him.

Working with Tom Leighton has been a pleasure. The things that he pulls from his bag of tricks continue to surprise me.

Besides Tom and Charles, several others made direct contributions to this thesis. Richard Koch, Satish Rao, and Arnold Rosenberg collaborated on various parts. I am particularly grateful to Satish for helping work out the details of one section even while he was writing his own thesis. Tom Cormen and James Park produced some of the figures, and Baruch Awerbuch graciously served as a thesis reader.

I have benefited from many technical discussions with my friends at MIT and elsewhere. Alok Aggarwal, Sanjeev Arora, Ravi Boppana, Ron Greenberg, Jon Greene, Michelangelo Grigni, Johan Hastad, Joe Kilian, James Park, Nick Pippenger, Abhiram Ranade, John Rompel, John Reif, Eric Schwabe, and Marc Snir all deserve mention.

During the past four years I have shared office NE43-313 with Tom Cormen, Sally Goldman, Alex Ishii, and Cliff Stein. I will miss the camaraderie and hubbub of the office.

The support staff at the Laboratory for Computer Science has always been helpful. Arline Benford and Be Hubbard kept me organized, and Sharon Thomas bent the rules for me.

Financial support was provided by an NSF Graduate Fellowship and by the Defense Advanced Research Projects Agency under Contract N00014-87-K-825.

Finally, I would like to thank my family and my wife Ginny for their support on the home-front.

Introduction

This thesis explores strategies for exploiting *locality* in parallel computation. Locality is perhaps best illustrated by the telephone system. A local phone call exhibits locality because it is transmitted over a small physical distance and through few switching stations. On the other hand, a long distance call may pass through many switching stations and span the globe. The telephone company exploits the aggregate locality of a typical set of phone calls by allocating more resources to local calls than to long distance calls. The communications hardware is arranged in a hierarchy, with bushy local networks at the bottom and a sparser satellite system at the top. The phone system itself may be said to exhibit locality in the sense that it reflects the locality of a typical set of calls.

The routing network in a parallel computer has a job much like that of the phone system. It must deliver packets of information between different processors. In this thesis, however, we shall restrict our attention to networks that are more tightly coupled than the phone system. These networks route packets to their destinations via a series of globally synchronized time steps. We model a routing network as a graph, where the nodes correspond to processors or switches, and the edges correspond to wires. At each step a packet can either traverse an edge or wait in a queue, and each edge can transmit at most one packet. The time to deliver a set of packets is the equal to the number of steps required for every packet to reach its destination.

Packet routing

Two important measures of the locality of a set of packets are its congestion and dilation. The *congestion*, c, of a set of packets is the maximum number of packets that use any edge of the network. The *dilation*, d, is the length of the longest path taken by any packet. Both of these

measures are lower bounds on the time required to deliver the messages. The congestion is a lower bound because c packets must pass through some edge, and at most one packet can traverse the edge at each time step. The dilation is a lower bound because some packet must travel a distance of d and it can travel a distance of at most one in each time step.

Chapter 1 describes a network-independent approach to the packet routing problem. Our strategy is to partition the problem into two stages: a path selection stage and a scheduling stage. The path-selection stage varies from network to network. Its goal is to find a set of paths for the packets that exhibits locality, i.e., has small congestion and dilation. The goal of the second stage is to determine when each packet should move, and when it should wait in a queue. The second stage must ensure that at most one packet traverses each edge at each time step. It should exploit the locality present in the paths produced by the first stage, i.e., the time to deliver the packets should be as close to the lower bounds c and d as possible and the queue size should be minimized.

The focus of Chapter 1 is on the second stage. Two main scheduling results are proved there. First we show that there is a schedule of length $O(c+d)$ for any set of packets with congestion c and dilation d (in any network) that uses constant-size queues. Unfortunately, no efficient algorithm for constructing it is known. However, for the special case of *layered* networks, we show that there is an efficient randomized algorithm for routing N packets in $O(c+d+\log N)$ steps using constant-size queues.

The algorithm for routing packets on layered networks has many applications to routing and sorting. Among them are the first on-line algorithms for routing N-packets on an N-node shuffle-exchange graph in $O(\log N)$ steps using constant-size queues and for routing kN packets on an N-node k-dimensional array with maximum side length M in $O(kM)$ steps using constant-size queues. The routing algorithm can also be used as a subroutine in sorting algorithms. It yields the first asymptotically optimal algorithms for sorting on butterfly, shuffle-exchange, and multidimensional array networks using constant-size queues.

A second major application area is in the construction of area-universal networks: N-node networks with VLSI-layout area $O(N)$ that can simulate all other networks with area $O(N)$ with only $O(\log N)$ slowdown. (The generalization to three dimensions is straightforward.) These networks are area-universal precisely because they display the kinds of locality present

in the phone system. The communications hardware is arranged in a hierarchy, with most of it devoted to making local connections.

Distributed random-access machines

Another important measure of locality is the *load factor* of a set of packets. Before defining the load factor, we need a few other notions. A *cut* S of a network is a subset of the nodes of the network. The capacity $\text{cap}(S)$ is the number of wires connecting processors in S to the rest of the network, $\overline{S}$. The *load* of a set M of packets on a cut S, $\text{load}(M, S)$, is the number of packets in M that must cross the cut S. The load factor of M on S, $\lambda(M, S)$ is the ratio of the load to the capacity, $\lambda(M, S) = \text{load}(M, S)/\text{cap}(S)$. The load factor of M on the entire network is the maximum load factor over all cuts, $\lambda(M) = \max_S \text{load}(M, S)$. The load factor is a lower bound on the congestion of any set of paths for the packets, and thus is a lower bound on the time to deliver the packets.

Chapter 2 introduces a model called the Distributed Random-Access Machine (DRAM) in which time required to deliver a set of packets is equal to its load factor. A DRAM is an abstraction of a parallel computer in which memory accesses are implemented by routing packets through a communication network. The model was originally intended to be an abstraction of a class of area-universal networks called fat-trees [29, 56]. Fat-trees are well modeled by DRAM's because, as we shall see in Chapter 1, the time to deliver a set M of packets on an N-node fat-tree is $O(\lambda(M) + \log N)$, with high probability.

The notion of load factor can be extended to measure the locality of a data structure embedded in a parallel computer. A natural way to embed a data structure in a DRAM is to put one record of the data structure into each processor. The record can contain data, including pointers to records in other processors. We measure the locality of an embedding by treating the data structure as a set of pointers and generalizing the concept of load factor to sets of pointers. The load of a set P of pointers across a cut S, denoted $\text{load}(P, S)$, is the number of pointers in P from a processor in S to a processor in $\overline{S}$, or vice versa. The load factor of P on the entire DRAM is $\lambda(P) = \max_S \text{load}(P, S)/\text{cap}(S)$. The load factor of a data structure is the load factor of the set of its pointers.

A *conservative algorithm* is a DRAM algorithm in which the load factor of the set of mem-

ory accesses produced at each step does not exceed the load factor of the input data structure. A conservative algorithm exploits the locality in the input data structure because it never produces more congestion across cuts of the DRAM than is implicit in the input data structure. Consequently, a conservative algorithm is guaranteed not to produce undue congestion in any underlying network. With the help of a lemma for "shortcutting" pointers in a data structure without increasing its load, we design fast conservative algorithms for a variety of graph problems. Problems such as computing treewalk numberings, finding the separator of a tree, and evaluating all subexpressions in an expression tree can be solved in $O(\log N)$ steps for N-node trees by conservative algorithms for an exclusive-read exclusive-write DRAM. More complex problems such as finding a minimum-cost spanning forest, computing biconnected components and constructing an Eulerian cycle require $O(\log^2 N)$ steps, for graphs of size N. For concurrent-read concurrent-write DRAM's, all of these problems can be solved by $O(\log N)$ step conservative algorithms.

Emulations

Of particular interest is the special case where the embedded data structure is a network. An *embedding* is a map from a guest network to a host network that takes nodes of the guest to nodes of the host, and edges of the guest to paths in the host. Three important measures of an embedding are its congestion, dilation, and load. The congestion and dilation of the paths are analogous to the congestion and dilation defined for the paths taken by a set of packets. The *load* of an embedding is the maximum number of guest nodes mapped to any one of the host nodes. The assignment of two meanings to the word load is unfortunate, but well established. In this thesis, the intended meaning should always be clear from the context. Furthermore, the load of a set M of packets on a cut S is denoted by $\text{load}(M, S)$, while the load of an embedding is denoted by l.

A guest network is typically embedded in a host network so that the host can *emulate* some computation to be performed by the guest. An important consequence of the scheduling results of Chapter 1 is that if a guest network can be embedded in a host network with load l, congestion c, and dilation d, then the host can emulate the guest with slowdown $O(l + c + d)$. Most of the efficient emulation schemes that we know of arise directly from an embedding of

a guest network in a host with small congestion, dilation, and load. As we shall see, however, a good embedding of the guest in the host is not required for the host to perform an efficient emulation of the guest.

Chapter 3 examines the problem of how efficiently a host network can emulate a guest network. The goal is to emulate T_G steps of an N_G-node guest network on an N_H node host network. We call an emulation *work-preserving* if the time required by the host, T_H is $O(T_G N_G/N_H)$ because then both the guest and host networks perform the same amount of total work (processor-time product), $\Theta(T_G N_G)$, to within a constant factor. A work-preserving emulation is efficient because it achieves optimal speedup over a sequential emulation of the guest. We say that an emulation is *real-time* if $T_H = O(T_G)$, because then the host emulates the guest with constant delay.

Although many isolated emulation results have been proved for specific networks in the past, and measures such as dilation and congestion were known to be important, the field has lacked a model within which general results and meaningful lower bounds could be proved. We attempt to provide such a model, along with techniques for proving lower bounds based on comparing the locality the networks. As a general rule, networks that exhibit locality are easier to emulate than those that do not.

The simplest measure of the locality of a network is its *diameter*. Let $\delta(u, v)$ denote the distance between a pair of nodes u and v, i.e., the length of the shortest path between u and v. The diameter, D, of a network is the maximum over all pairs (u, v) of the distance between u and v, $D = \max_{(u,v)} \delta(u, v)$. In general, a network with large diameter exhibits more locality than a network with small diameter. For example, a linear array exhibits more locality than a shuffle-exchange graph.

The *expansion rate* is another important measure of the locality of a network. Let $B_r(u)$ denote the ball of radius r around a node u, i.e., the set of nodes within distance r of u, $B_r(u) = \{v | \delta(u, v) \leq r\}$. For a set S of nodes, the *neighborhood* of S, $N(S)$, is the set of nodes within a distance of 1 of some node in S, excluding those nodes in S, $N(S) = \cup_{u \in S} B_1(u)) - S$. We say that an n-node network has expansion rate ε if for every set S of size at most $n/2$, the size of the neighborhood of S is a least $\varepsilon|S|$. We call a network for which the expansion rate ε is at least some fixed positive constant an expander. An expander exhibits little locality.

Some of the more interesting and diverse results in Chapter 3 include a proof that a linear array can emulate a (much larger) butterfly in a work-preserving fashion, but that a butterfly cannot emulate an expander (of any size) in a work-preserving fashion; a proof that a mesh can be emulated in real time in a work-preserving fashion on a butterfly, even though any $O(1)$-to-1 embedding of the mesh in a butterfly has dilation $\Omega(\log N)$; and a proof that an N-node butterfly can emulate an $N \log N$-node shuffle-exchange graph in a work-preserving fashion, and vice-versa.

Mesh-based algorithms

Chapter 4 presents an algorithm for finding a minimum-cost spanning tree of an N-node graph on an $N \times N$ mesh-connected computer. The algorithm has the same $O(N)$ running time as the previous algorithms, but it is much simpler. In VLSI models, the mesh is the ultimate local network because each processor in the mesh is connected to a small number of neighbors by minimum length wires.

Chapter 1

Packet routing algorithms

1.1 Introduction

Figure 1-1 illustrates the standard graph model for packet routing. The shaded nodes labeled 1 through 5 represent processors or switches. The edges between the nodes represent wires. At the end of each edge is an *edge queue* that can hold a small number of packets (in this example, two). A packet is depicted by a square box containing the label of its destination. Before the routing begins, packets are stored at their origins in special *initial queues*. For example, packets 4 and 5 are stored in the initial queue at node 1.

The goal is to route each packet from its origin to its destination via a series of synchronized time steps. At each step at most one packet can traverse each edge. Furthermore, a packet can traverse an edge only if at the beginning of the step its edge queue is not full. Upon traversing the last edge on its path, a packet is removed from the edge queue placed in a special *final queue* at its destination. For simplicity, the final queues are not shown in Figure 1-1. Independent of the routing algorithm used, the size of the initial and final queues are determined by the particular packet routing problem to solved. Thus, any bound on the maximum queue size required by a routing algorithm refers to the edge queues only.

The task of designing an efficient packet routing algorithm is central to the design of most large-scale general-purpose parallel computers. In fact, even the basic unit of time in some parallel machines is measured in terms of how fast the packet router operates. For example,

This chapter describes joint research with Tom Leighton and Satish Rao [53].

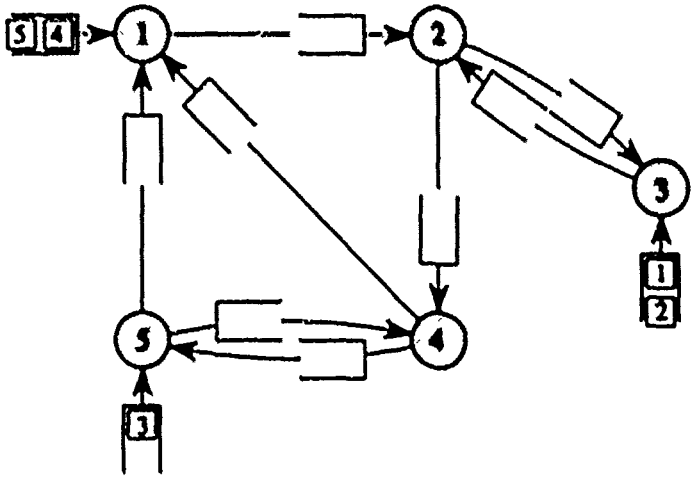

Figure 1-1: A graph model for packet routing.

the speed of an algorithm in the Connection Machine is often measured in terms of *routing cycles* (roughly the time to route a random permutation) or *petit cycles* (the time to perform an atomic step of the routing algorithm). Similarly, the performance of machines like the BBN Butterfly is substantially influenced by the speed and rate of successful delivery of its router.

Packet routing also provides an important bridge between theoretical computer science and applied computer science; it is through packet routing that a real machine such as the Connection Machine is able to simulate an idealized machine such as the CRCW PRAM. More generally, getting the right data to the right place at the right time is an important, interesting, and challenging problem. Not surprisingly, it has also been the subject of a great deal of research.

1.1.1 Past work

The first major result in packet routing is due to Benes [10] who showed that the inputs and outputs of a Benes network can be connected in any permutation by a set of disjoint paths. Waksman [98] then gave a simple off-line algorithm for finding the paths in linear time. Given the paths, it is straightforward to route a permutation of packets from the inputs to the outputs of an N-node Benes network in $O(\log N)$ steps using queues of size 1. Although the inputs comprise only $O(N/\log N)$ nodes, it is possible to route any permutation of N packets

in $O(\log N)$ steps by pipelining $\Theta(\log N)$ such permutations. Unfortunately, no efficient on-line algorithm for finding the paths is known.

Shortly thereafter, Batcher [9] devised an elegant and practical on-line algorithm for sorting N packets on an N-node shuffle-exchange graph in $\log^2 N$ steps using queues of size 1. The algorithm can be used to route any permutation of packets by sorting based on destination address. The result extends to routing many-one problems provided that (as is typically assumed) combining can be used to merge packets that have a common destination.

No better deterministic algorithm was found until Ajtai, Komlos, and Szemeredi [2] solved a classic open problem by constructing an $O(\log N)$-depth sorting network. Leighton [47] then used this $O(N \log N)$-node network to construct a degree 3 N-node network capable of solving any N-packet routing problem in $O(\log N)$ steps using queues of size 1. Although this result is optimal up to constant factors, the constant factors are quite large and the algorithm is of no practical use. Hence, the effort to find fast deterministic algorithms has continued. Recently Upfal discovered an $O(\log N)$-step algorithm for routing on an expander-based network called the multibutterfly [95]. The algorithm solves the routing problem directly without reducing it to sorting, and the constant factors are much smaller than those of the AKS-based algorithms. In [52], we show that the multibutterfly is fault tolerant and improve the constant factors in Upfal's algorithm.

There has also been great success in the development of efficient randomized packet routing algorithms. The study of randomized algorithms was pioneered by Valiant and Brebner [97] who showed how to route any permutation of N packets in $O(\log N)$ steps on an N-node hypercube with queues of size $O(\log N)$ at each node. Although the algorithm is not always guaranteed to work, it is guaranteed to work with probability at least $1 - 1/N$ for any permutation. This result was improved in a succession of fundamental papers by Aleliunas [3], Upfal [94], Pippenger [76], and Ranade [81]. Aleliunas and Upfal developed the notion of a *delay path* and showed how to route on the shuffle-exchange and butterfly graphs (respectively) in $O(\log N)$ steps with queues of size $O(\log N)$. Pippenger was the first to eliminate the need for large queues, and showed how to route on a variant of the butterfly in $O(\log N)$ steps with queues of size $O(1)$. Ranade showed how combining could be used to extend the Pippenger result to include many-one routing problems, and tremendously simplified the analysis required to prove

such a result. As a consequence of Ranade's work, it has finally become possible to simulate a step of an N-processor CRCW PRAM on an N-node butterfly or hypercube in $O(\log N)$ steps using constant-size queues on each edge.

Concurrent with the development of these hypercube-related packet routing algorithms has been the development of algorithms for routing in meshes. The randomized algorithm of Valiant and Brebner can be used to route any permutation of N packets on a $\sqrt{N} \times \sqrt{N}$ mesh in $O(\sqrt{N})$ steps using queues of size $O(\log N)$. Kunde [43] showed how to route deterministically in $(2+\varepsilon)\sqrt{N}$ steps using queues of size $O(1/\varepsilon)$. Also, Krizanc, Rajasekaran, and Tsantilis [41] showed how to randomly route any permutation in $2\sqrt{N} + O(\log N)$ steps using constant size queues. Most recently, Leighton, Makedon, and Tollis discovered a deterministic algorithm for routing any permutation in $2\sqrt{N} - 2$ steps using constant-size queues [49], thus achieving the optimal time bound in the worst case.

1.1.2 Our approach

One deficiency with the state-of-the-art in packet routing is that aside from Valiant's paradigm of "first routing to a random destination," all of the algorithms and their analyses are very specifically tied to the network on which the routing is to take place, as well as to the requirement that packets are first routed to destinations that are (in some sense) random. For example, the butterfly routing algorithms are all quite different than the mesh algorithms in the way that queue size is kept constant. Moreover, the butterfly and hypercube algorithms are so specific to those networks that no $O(\log N)$-step constant-queue-size algorithm was known for the closely related shuffle-exchange graph. The lack of a good routing algorithm for the shuffle-exchange graph is one of the reasons that the butterfly is preferred to the shuffle-exchange graph in practice.

In this chapter, we take a significant step towards the development of a universal approach to packet routing. Our approach to the problem differs from previous approaches in that we separate the process of selecting packet paths from the process of timing packet movements along the paths. More precisely, given any underlying network, and any selection of paths for the packets, we study the problem of timing the movement of the packets so as to minimize the total time and maximum queue size needed to route all the packets to their correct destinations.

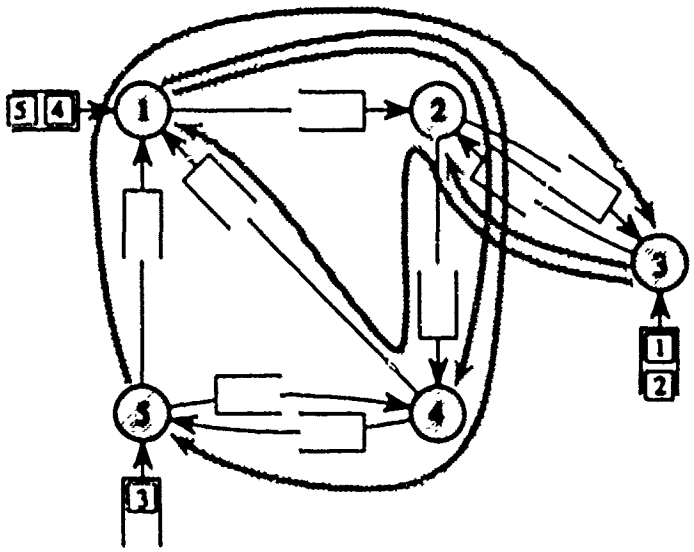

Figure 1-2: A set of paths for the packets. Each packet follows a shortest path to its destination. The dilation is $d = 3$ and the congestion is $c = 3$.

Of course, there must be some correlation between the performance of the algorithm and the selection of the paths. In particular, the maximum distance, d, traveled by any packet is always a lower bound on the time required to route all packets. We call this distance the *dilation* of the paths. Similarly, the largest number of packets that must traverse a single edge during the entire course of the routing is a lower bound. We call this number the *congestion*, c, of the paths.

Viewed in terms of these parameters, then, a routing problem can be broken into two stages. In Stage 1, we select paths for the packets so as to minimize c and d. In Stage 2, we schedule the movement of the packets so as to minimize the total time and maximum queue size. To illustrate this two stage approach, let us return to the routing problem of Figure 1-1.

Figure 1-2 shows one way of choosing the paths for the packets. Here, each packet takes a shortest path from its origin to its destination. For example, packet 1 follows a path from node 3 to 2 to 4 to 1. Since no packet traverses more than three edges, the dilation is $d = 3$. Packets 3, 4, and 5 all traverse the edge from 1 to 2, but no more than three packets traverse any other edge. Thus, the congestion is $c = 3$.

A schedule for the packets is displayed in Figure 1-3. A schedule simply specifies which

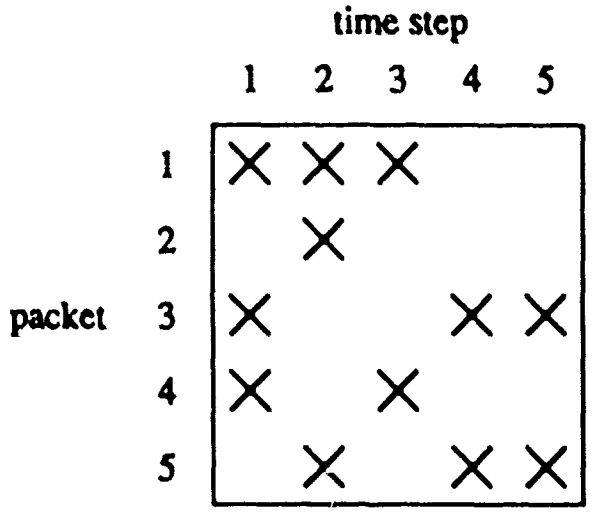

Figure 1-3: A schedule for the packets. An × in row p and column t indicates that at time t packet p moves. A blank indicates that it waits.

packets move and which wait at each time step. An × in row p and column t indicates that at time t packet p traverses an edge and enters the queue at the end of that edge. A blank indicates that at time t packet p waits in a queue. For example, packet 3 moves at time step 1, waits at steps 2 and 3, and then moves again in steps 4 and 5.

The step-by-step progress of the packets as they follow the paths from Figure 1-2 according to the schedule of Figure 1-3 is illustrated in Figure 1-4.

Part (a) shows the packets in their initial queues before the routing begins. In the first step, packet 1 takes the edge from node 3 to node 2, 3 takes the edge from 5 to 1, and 4 takes the edge from 1 to 2. Packets 2 and 5 must wait because the first edges on their paths are taken by packets 1 and 4, respectively.

The positions of the packets at the end of time step 1 are shown in part (b). In step 2, packets 1, 2, and 5 move, while packets 3 and 4 wait. Packet 2 reaches its destination, is removed from the queue at the end of the edge from 3 to 2, and enters the final queue for node 2.

At the end of step 2, the packets are positioned as shown in part (c). Note that packet 2, which resides in the final queue for node 3, is not pictured. In step 3, packets 1 and 4 move,

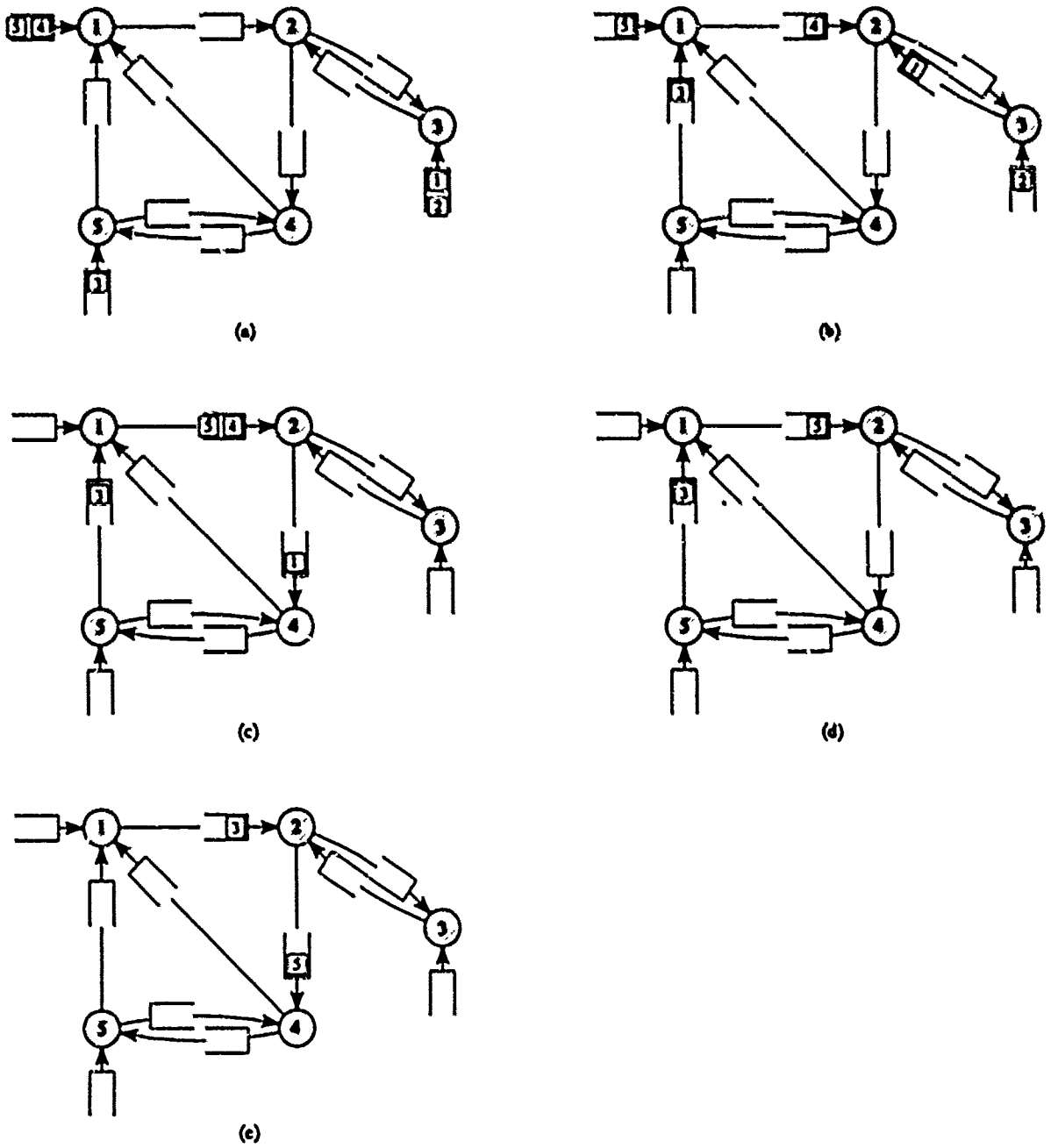

Figure 1-4: The step-by-step progress of the packets. The positions of the packets at the ends of steps 0 through 4 are shown in parts (a) through (e) respectively.

but packet 3 must wait because the queue that it wishes to enter is full at the beginning of the step.

After step 3, only packets 3 and 5 remain en route. Both packets move in step 4, and reach their destinations in step 5. Their positions at the ends of steps 3 and 4 are show in parts (d) and (e), respectively.

For many networks, Stage 1 is easy. We simply use Valiant's paradigm of first routing to a random destination, and then routing to the correct destination. It is easily shown for meshes, butterflies, shuffle-exchange graphs, etc., that this approach yields values of c and d that are within a small constant factor of the diameter of the network, which is as well as can be done. Moreover, this technique also usually works for many-one problems provided that the address space is randomly hashed.

Stage 2 has traditionally been the hard part of routing. Curiously, however, we have found that by ignoring the underlying network and the method of path selection, Stage 2 actually becomes easier to solve! Hence we will be able to obtain results for routing that are both simpler and far more general than existing approaches. Among other things, we will be able to route on the N-node mesh in $O(\sqrt{N})$ steps using constant size queues with the same algorithm that uses $O(\log N)$ steps and constant-size queues on the butterfly. We will also be able to route on the shuffle-exchange graph in $O(\log N)$ steps with constant-size queues. Also, by showing how to route efficiently on a fat-tree, we provide the first examples of volume and area-universal networks that require only $O(\log N)$ slowdown.

1.1.3 Outline of the results

Our most difficult result is a proof that any set of packets whose paths have congestion c and dilation d can be scheduled so as to complete the routing in $O(c+d)$ steps using constant-size queues. This result is optimal up to constant factors, and substantially improves the naive bound of $O(cd)$ steps and $O(c)$ size queues. Unfortunately, the result is highly nonconstructive, and therefore is useful only if substantial amounts of off-line computation are available for the routing. On the other hand, the result is robust in the sense that it provides near-optimal schedule of packet movements for any set of paths and any underlying network. Such robustness is particularly useful when dealing with routing problems on arbitrary distributed networks as

in [54]. The proof of the result is contained in Section 1.2.

We do not know whether or not there is an on-line algorithm that can route any set of paths in $O(c+d)$ steps with constant-size queues. It is not difficult to devise a randomized on-line algorithm to schedule any set of N paths in $O(c+d\log N)$ steps using queues of size $O(\log N)$. In special cases, however, we can do better. For example, a slight variant of Ranade's algorithm can be used to schedule on-line any set of N paths on a bounded-degree layered network in $O(c+d+\log N)$ steps using constant-size queues. By a *layered* network, we mean a network in which each edge connects a level i node to a level $i+1$ node, where the level numbers range from 0 to d. For example, the butterfly is layered this fashion. The algorithm is randomized, but requires only $\Theta(\log^2 N)$ bits of randomness to succeed with high probability. The proof of this result is included in Section 1.3. Curiously, the proof is simpler than the previous proof of the same result applied specifically to routing random paths in butterflies [81]. (The fact that Ranade's algorithm can be used in this general context has also been observed by Ranade [82].)

The on-line algorithm for layered networks can immediately be applied to obtain good routing algorithms for meshes and butterflies. With some extra effort, it can be applied to obtain the first algorithm for routing kN packets on an N-node k-dimensional array with maximum side length M in $O(kM)$ steps. constant-size queues, and for routing N-packets on an N-node shuffle-exchange graph in $O(\log N)$ steps using constant-size queues. It can also be applied to construct a class of networks that are *area universal* in the sense that the network in the class with N processors has area $O(N)$, and can, with high probability, simulate in $O(\log N)$ steps each step of any other network of area $O(N)$. An analogous result is shown for a class of *volume-universal* networks. The routing algorithm is used as a subroutine in algorithms for sorting on butterflies and multidimensional arrays. The details of these applications are included in Sections 1.4 through 1.9.

This thesis leaves open the question of whether or not there is an on-line algorithm that can schedule any set of paths in $O(c+d)$ steps using constant-size queues. We suspect that finding such an algorithm (if one exists) will be a challenging task. Our negative suspicions are derived from the fact that we can construct counterexamples to most of the simplest on-line algorithms. In other words, for several natural on-line algorithms (including the algorithm described in Section 1.3) we can find packet paths for which the algorithm will construct a

schedule using substantially more than $\Omega(c+d+\log N)$ steps. Several of the counterexamples are included in Section 1.10.

1.2 The existence of asymptotically optimal schedules

The main result in this section is a proof that for any set of packets whose paths are edge-simple[1] and have congestion c and dilation d, there is a schedule of length $O(c+d)$ in which at most one packet traverses each edge of the network at each step, and at most $O(1)$ packets wait in each queue at each step. Note that there are no restrictions on the size, topology, or degree of the network or on the number of packets.

Our strategy for constructing an efficient schedule is to make a succession of refinements to the "greedy" schedule, S_0, in which each packet moves at every step until it reaches its final destination. This schedule is as short as possible; its length is only d. Unfortunately, as many as c packets may use an edge at a single time step in S_0, whereas in the final schedule at most one packet is allowed to use an edge at each step. Each refinement will bring us closer to meeting this requirement by bounding the congestion within smaller and smaller frames of time.

The proof uses the Lovasz local lemma [89, pp. 57–58] at each refinement step. Given a set of "bad" events in a probability space, the lemma provides a simple inequality which when satisfied guarantees that with probability greater than zero, no bad event occurs. The inequality relates the probability that each bad event occurs with the dependence among them. A set of events $A_1,\ldots,A_m$ in a probability space has *dependence* at most b if every event is mutually independent of some set of $m-b$ other bad events. The lemma is nonconstructive; for a discrete probability space it proves that there is some elementary outcome that is not in any bad event, but does not specify that outcome.

Lemma 1 (Lovasz) *Let $A_1,\ldots,A_m$ be a set of "bad" events each occurring with probability p with dependence at most b. If $4pb < 1$, then with probability greater than zero, no bad event occurs.* □

[1] An *edge-simple* path uses no edge more than once.

1.2.1 A preliminary result

Before proving the main result of this section, we show that there is a schedule of length $(c+d)2^{O(\log^*(c+d))}$ that uses queues of size $\log(c+d)2^{O(\log^*(c+d))}$. This preliminary result is substantially simpler to prove because of the relaxed bounds on the schedule length and queue size. Nevertheless, it illustrates the basic ideas necessary to prove the main result.

Theorem 2 *For any set of packets whose paths are edge-simple and have congestion c and dilation d, there is a schedule in which at most one packet traverses each edge at each step with length $(c+d)2^{O(\log^*(c+d))}$ and maximum queue size $\log(c+d)2^{O(\log^*(c+d))}$.*

Proof: For simplicity, we shall assume without loss of generality that $c = d$, so that the bounds on the length and queue size are $d2^{O(\log^* d)}$ and $(\log d)2^{O(\log^* d)}$, respectively.

The proof has the following outline. The first step is to assign each packet a delay chosen randomly, independently, and uniformly from the range $[1, \alpha d]$, where α is a fixed constant that will be determined later. In the resulting schedule, S_1, a packet assigned a delay of x waits in its initial queue for x steps, then moves on to its destination without waiting again until it enters is final queue. The length of S_1 is at most $(1+\alpha)d$. Next we break the schedule into $(1+\alpha)d/\log d$ sets of $\log d$ consecutive time steps, as shown in Figure 1-5. Each of these sets is called a $\log d$-*frame*. We use the Lovasz local lemma to show that there is some way of choosing the initial delays so that in each of these $\log d$-frames at most $\log d$ packets pass through any edge. Finally, we view each $\log d$-frame as a routing problem with dilation $\log d$ and congestion $\log d$, and solve it recursively.

To apply the Lovasz Local Lemma, we associate a bad event with each edge. The bad event for edge e is that more than $\log d$ packets use e in any $\log d$-frame. To show that there is a way of choosing the delays so that no bad event occurs, we need to bound the dependence, b, among the bad events and the probability, p, of each individual bad event occurring.

The dependence calculation is straightforward. Whether or not a bad event occurs depends solely on the delays assigned to the packets that pass through the corresponding edge. Thus, two bad events are independent unless some packet passes through both of the corresponding edges. Since at most $c = d$ packets pass through an edge, and each of these packets passes through at most d other edges, the dependence, b, of the bad events is at most $cd = d^2$.

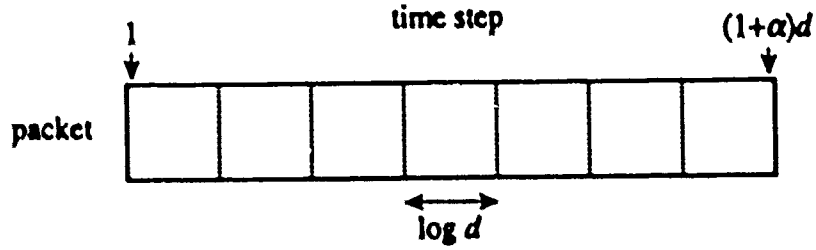

Figure 1-5: Schedule S_1. The schedule is derived from the greedy schedule, S_0, by assigning each packet a random initial delay in the range $[1, \alpha d]$. We use the Lovasz local lemma to show that within each $\log d$-frame, at most $\log d$ packets pass through each edge.

Computing the probability of each bad event is a little trickier. Let p be the probability of the bad event corresponding to edge e. Then

$$p \leq \frac{(1+\alpha)d}{\log d}\binom{d}{\log d}\left(\frac{\log d}{\alpha d}\right)^{\log d}.$$

This expression is derived as follows. There are $(1+\alpha)d/\log d$ different $\log d$-frames, and we bound p by summing over all frames the probability that more than $\log d$ packets pass through e in the frame. The number of packets passing through e in the frame has a binomial distribution. There are d independent Bernoulli trials, one for each packet that uses e. Since at most $\log d$ of the possible αd delays will actually send a packet through e in the frame, each trial succeeds with probability $\log d/\alpha d$. (Here we use the assumption that the paths are edge-simple.) The probability of more than $\log d$ successes is at most $\binom{d}{\log d}\left(\frac{\log d}{\alpha d}\right)^{\log d}$.

For sufficiently large α, the product $4pb$ is less than 1, and thus, by the Lovasz Local Lemma, there is some assignment of delays such that at most $\log d$ packets use any edge in any $\log d$-frame.

Each $\log d$-frame can be viewed as a separate scheduling problem where the origin of a packet is its location at the beginning of the frame, and its destination is its location at the end of the frame. If at most $\log d$ packets use each edge in a $\log d$-frame, then the congestion of the problem is $\log d$. The dilation is also $\log d$ because in $\log d$ time steps a packet can move a distance of at most $\log d$. In order to schedule each frame independently, a packet that arrives at its destination before the last step in the rescheduled frame is forced to wait there until the

next frame begins.

All that remains is to bound the length of the schedule and the size of the queues. The recursion proceeds to a depth of $O(\log^* d)$ at which point the frames have size $O(1)$, and at most $O(1)$ packets use each edge in each frame. The resulting schedule can be converted to one in which at most one packet uses each edge in each time step by slowing it down by a constant factor. The length of the final schedule is $d2^{O(\log^* d)}$. The bound on the queue size follows from the observation that no packet waits at any one spot (other than its origin or destination) for more than $(\log d)2^{O(\log^* d)}$ consecutive time steps, and in the final schedule at most one packet traverses each edge at each time step. □

1.2.2 The main result

Proving that there is a schedule of length $O(c+d)$ using constant-size queues is more difficult. Removing the $2^{O(\log^*(c+d))}$ factor in the length of the schedule seems to require delving into second order terms in the probability calculations, and reducing the queue size to $O(1)$ mandates greater care in spreading delays out over the schedule.

Before proceeding, we need to introduce some notation. The *frame congestion*, C, in a T-frame is the largest number of packets that traverse any edge in the frame. The *relative congestion*, R, in a T-frame is the ratio C/T of the congestion in the frame to the size of the frame.

Theorem 3 *For any set of packets whose paths are edge-simple and have congestion c and dilation d, there is a schedule in which at most one packet traverses each edge of the network at each step with length $O(c+d)$ and maximum queue size $O(1)$.*

Proof: To make the proof more modular, bounds on frame size and relative congestion after each step in the construction are stated as lemmas. These lemmas and their proofs are included within the proof of the theorem. We assume without loss of generality that $c = d$, so that the bound on the length of the schedule is $O(d)$.

As before, the strategy is to make a succession of refinements to the greedy schedule, S_0. The first refinement is special. It transforms S_0 into a schedule S_1 in which the relative congestion in each $\log d$-frame is at most $O(1)$. Thereafter, each refinement transforms a schedule S_i with

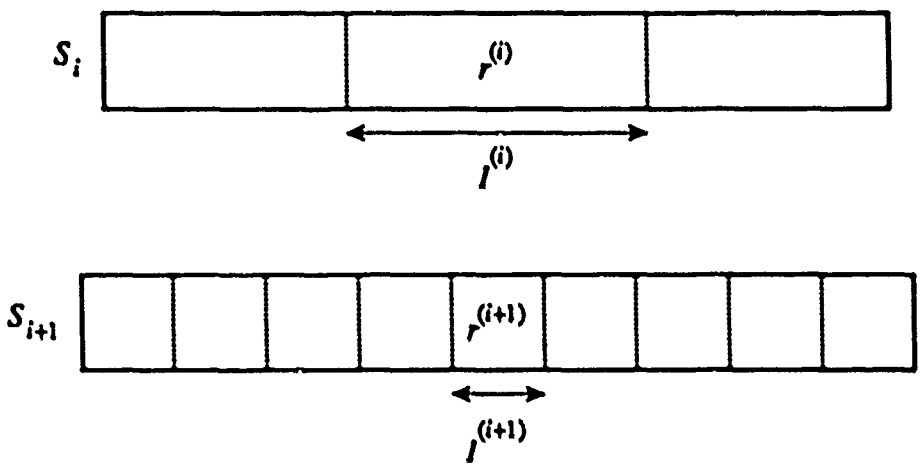

Figure 1-6: A refinement step. Each refinement transforms a schedule S_i into a slightly longer schedule S_{i+1}. The frame size is greatly reduced in S_{i+1}, yet the relative congestion within a frame remains about the same, i.e., $I^{(i+1)} \ll I^{(i)}$ and $r^{(i+1)} \approx r^{(i)}$.

relative congestion at most $r^{(i)}$ in any frame of size $I^{(i)}$ or greater into a schedule S_{i+1} with relative congestion at most $r^{(i+1)}$ in any frame of size $I^{(i+1)}$ or greater, where $r^{(i+1)} \approx r^{(i)}$ and $I^{(i+1)} \ll I^{(i)}$, as shown in Figure 1-6. As well shall see, after j refinements, where $j = O(\log^* d)$, we obtain a schedule S_j with relative congestion $O(1)$ in every frame of size k_0 or greater, where k_0 is some constant. From S_j it is straightforward to construct a schedule of length $O(c+d)$ in which at most one packet traverses each edge of the network at each step, and at most $O(1)$ packets wait in each queue at each step.

At the start, the relative congestion in a d-frame of S_0 is at most 1. We begin by assigning each packet a random delay chosen uniformly from 1 to d at the beginning of the greedy schedule S_0. Using the Lovasz local lemma, it is possible to show that there is some way of choosing the delays so that in the resulting schedule S_1, the relative congestion is at most $r^{(1)} = O(1)$ in any frame of size $I^{(1)} = \log d$ or greater.

Next, we repeatedly refine the schedule to reduce the frame size. As we shall see, the relative congestion $r^{(i+1)}$ and frame size $I^{(i+1)}$ for schedule S_{i+1} are given by the recurrences

$$r^{(i+1)} = \begin{cases} O(1) & i = 1 \\ r^{(i)}(1 + O(1)/\sqrt{\log I^{(i)}}) & i > 1 \end{cases}$$

and

$$I^{(i+1)} = \begin{cases} \log d & i = 1 \\ \log^4 I^{(i)} & i > 1 \end{cases}$$

which have solutions $I^{(j)} = O(1)$ and $r^{(j)} = O(1)$ for some j, where $j = O(\log^* d)$.

We have not explicitly defined the values of $r^{(i)}$ and $I^{(i)}$ for which the recursion terminates. However, in several places in the proof that follows we implicitly use the fact that $I^{(i)}$ is sufficiently large or $r^{(i)}$ is sufficiently small that some inequality holds. The recursion terminates when the first of these inequalities fails to hold. When this happens, one of $r^{(i)}$ or $I^{(i)}$ is $O(1)$, which implies that the other is also.

An important invariant that we main maintain throughout the construction is that in schedule S_{i+1} every packet waits at most once every $I^{(i)}$ steps. As a consequence, a packet waits at most once every $\Omega(1)$ steps in S_j, which implies both that the queues in S_j cannot grow larger than $O(1)$ and that the total length of S_j is $O(d)$. Schedule S_j almost satisfies the requirement that at most one packet traverse each edge in each step. By simulating each step of S_j in $O(1)$ steps we can meet this requirement with only a factor of 2 increase in the queue size and a factor of $O(1)$ increase in the running time.

The rest of the proof describes a refinement step in detail. For ease of notation, we use I and r in place of $I^{(i)}$ and $r^{(i)}$.

The first step in the ith refinement is to break schedule S_i into *blocks* of $2I^3 + 2I^2 - I$ consecutive time steps. Each block is rescheduled independently.

For each block, each packet is assigned a random delay chosen independently and uniformly from 1 to I. A packet assigned a delay of x must wait for x steps at the beginning of the block. In order maintain the invariant that in schedule S_{i+1} every packet waits at most once every $I^{(i)}$ steps, the packet is not delayed for x consecutive steps at the beginning of the block, but instead a delay is inserted every I steps in the first xI steps of the block.[2] A packet that is delayed x steps reaches its destination at the end of the block by step $2I^3 + 2I^2 - I + x$. Since some packet may have delay $x = I$, the rescheduled block must have length $2I^3 + 2I^2$.

[2]Before the delays for schedule S_{i+1} have been inserted, a packet is delayed at most once in each block of S_i. Prior to inserting each new delay into a block, we check if it is within $I^{(i)}$ steps of the single old delay. If the new delay would be too close to the old delay, then it is simply not inserted. The loss of a single delay in a block has a negligible effect on the probability calculations in the lemmas that follow.

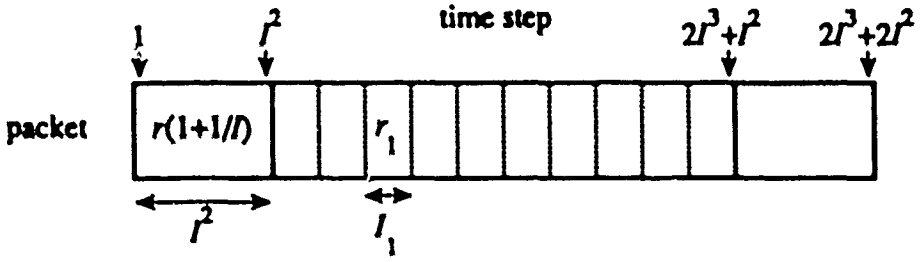

Figure 1-7: Bounds on frame size and relative congestion after inserting delays into S_i. Here $I_1 = \log^2 I$ and $r_1 = r(1 + O(1)/\sqrt{\log I})$.

In order to independently reschedule the next block, the packets must reside in exactly the same queues at the end of the rescheduled block that they did at the end of the block of S_i. Since some packets arrive early, they must be slowed down. Thus, if a packet is assigned delay x, then a delay is inserted every I steps in the last $I(I-x)$ steps of the block. Note that at the beginning of the first block and end of the last block, it is not necessary to separate the delays by I steps, because the packets reside in their initial and final queues, respectively.

Lemmas 4 through 6 bound the frame size and relative congestion in various parts of the block after the delays are inserted into S_i. The bounds are shown in Figure 1-7. Inserting delays may increase the relative congestion in the I^2 steps at the beginning and end of each block. Lemma 4 shows that by increasing the frame size from I to I^2 we can bound the relative congestion in these regions by $r(1 + 1/I)$. Lemma 6 shows that between the first and last I^2 steps we can decrease the frame size from I to $\log^2 I$, while only increasing the relative congestion in each frame from r to $r(1 + O(1)/\sqrt{\log I})$. The proof of Lemma 6 uses Lemma 5 to bound the relative congestion over a wide range of frame sizes.

Lemma 4 *For any choice of delays, the relative congestion in any frame of size I^2 or greater after the delays are inserted is at most $r(1 + 1/I)$.*

Proof: After the delays are inserted, a packet can use an edge in a T-frame if it used the edge in the frame or in any of the I steps before the frame in S_i. Thus, at most $r(T + I)$ packets can use an edge in the T-frame. For $T \geq I^2$, the relative congestion is at most $r(1 + 1/I)$. □

Lemma 5 *In any schedule, if the relative congestion in every frame of size T to $2T - 1$ is at*

most R then the relative congestion in any frame of size T or greater is at most R.

Proof: Consider a frame of size T', where $T' > 2T - 1$. The first $(\lfloor T'/T \rfloor - 1)T$ steps of the frame can be broken into T-frames, each with relative congestion R. The remainder of the T'-frame consists of a single frame of size between T and $2T - 1$ steps in which the relative congestion is also at most R. □

Lemma 6 *There is some way of choosing the packet delays so that in between the first and last I^2 steps of a block, the relative congestion in any frame of size $I_1 = \log^2 I$ or greater is at most $r_1 = r(1 + \epsilon_1)$, where $\epsilon_1 = O(1)/\sqrt{\log I}$.*

Proof: With each edge we associate a bad event. For edge e, a bad event occurs when more than $r_1 T$ packets use e in any T-frame for T in the range I_1 to $2I_1 - 1$. To show that no bad event occurs, we need to bound both the dependence of the bad events and the probability that an individual bad event occurs.

We first bound the dependence. At most $r(2I^3 + 2I^2 - I)$ packets use an edge in the block[3]. Each of these packets travels through at most $2I^3 + 2I^2 - I$ other edges in the block. As we shall see later, it will always be true that $r = r^{(i)} = O(1)$. Thus a bad event depends on $b = O(I^6)$ other bad events.

Now let us compute an upper bound on the probability, p_1, that more than $r_1 I_1$ packets use an edge in a particular I_1-frame. Since a packet may be delayed up to I steps before the frame, any packet that uses e in the frame or in any of the I steps before the frame in S_i may use e after the delays are inserted into S_i. Thus, there are at most $r(I + I_1)$ packets that can use e in the frame. For each of these the probability that the packet uses e in the frame after being delayed is at most (I_1/I). If we assume that no packet uses an edge more than once, then these probabilities are independent. Thus, the probability p_1 that more than $r_1 I_1$ packets use the frame is at most

$$p_1 \leq \sum_{k=r_1 I_1}^{r(I+I_1)} \binom{r(I + I_1)}{k} (I_1/I)^k (1 - I_1/I)^{r(I+I_1)-k}.$$

[3]Throughout the following lemmas we make references to quantities such as rI packets or $\log^4 I$ time steps, when in fact rI and $\log^4 I$ may not be integral. Rounding these quantities to integer values when necessary does not affect the correctness of the proof. For ease of exposition, we shall henceforth cease to consider the issue.

Let $r_1 = r(1+\varepsilon_1)$. We bound the series as follows. There are at most $r(I+I_1)$ terms, and the largest of these occurs for $k = r_1 I_1$. Applying the inequalities $(1+x) \leq e^x$, $\ln(1+x) \geq x - x^2/2$ for $0 \leq x \leq 1$, and $\binom{a}{b} \leq (ae/b)^b$ for $0 < b < a$ to this term, we have

$$p_1 \leq r(I+I_1)e^{-rI_1\varepsilon_1^2(1/2-\varepsilon_1/2-I_1/\varepsilon_1^2 I-2I_1/\varepsilon_1 I)}.$$

For $I_1 = \log^2 I$ and $\varepsilon_1 = k_1/\sqrt{\log I}$, we can ensure that $p \leq 1/I^{k_2}$, for any constant $k_2 > 0$ by making constant k_1 large enough.

Next we need to bound the probability p_2 that more than $r_1 I_1$ packets use e in any I_1-frame of the block. There are at most $O(I^3)$ I_1-frames. Thus $p_2 \leq O(I^3)p_1$. By making the constant k_2 large enough, we can ensure that $p_2 \leq 1/I^{k_3}$, for any constant $k_3 > 0$.

The calculations for frames of size $I_1 + 1$ through $2I_1 - 1$ are similar. There are at most $O(I^3)$ frames of any one size, and $2I_1$ frame sizes between I_1 and $2I_1 - 1$. By adjusting the constants as before, we can guarantee that the probability p that more than $r_1 T$ packets use e in any T-frame for T between I_1 and $2I_1 - 1$ is at most $1/I^{k_4}$ for any constant $k_4 > 0$.

Finally, since a bad event depends on only $b = O(I^6)$ other bad events, we can make $4pb < 1$ by making k_4 large enough. By the Lovasz local lemma, there is some way of choosing the packet delays so that no bad event occurs. □

Although the frame size in the center of each block has decreased, it has increased from I to I^2 in the first and last I^2 steps of the block. Before decreasing the frame size in these regions, we move the block boundaries to the centers of the blocks, as shown in Figure 1-8. Now each block of size $2I^3 + 2I^2$ has a "fuzzy" region of size $2I^2$ in its center in which the relative congestion in any frame of size I^2 or greater is $r(1 + 1/I)$. In the I^3 steps before and after the fuzzy region, the relative congestion in any frame of size I_1 or greater is r_1. To reduce the frame size in the fuzzy region, we assign a random delay from 1 to I^2 to each packet. A packet with delay x waits once every I^3/x steps in the I^3 steps before the fuzzy region and once every $I^3/(I^2 - x)$ steps in the I^3 steps after the region. The rescheduled block now has size $2I^3 + 3I^2$.

We now show that there is some way of inserting delays into the schedule before the fuzzy region that both reduces the frame size in the fuzzy region, and does not increase either the frame size or the relative congestion before the fuzzy region by much. A similar analysis holds after the fuzzy region.

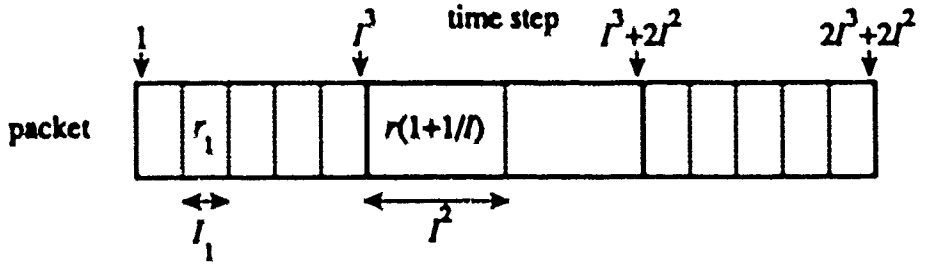

Figure 1-8: A block after recentering. The "fuzzy region" in the center of the block is shaded.

Lemma 7 *There is some way of choosing the packet delays so that between steps $I\log^3 I$ and steps I^3, the relative congestion in any frame of size I_1 or greater is at most $r_2 = r(1+\varepsilon_2)$, where $\varepsilon_2 = O(1)/\sqrt{\log I}$, and so that in the fuzzy region the relative congestion in any frame of size I_1 or greater is at most $r_3 = r(1+\varepsilon_3)$, where $\varepsilon_3 = O(1)/\sqrt{\log I}$.*

Proof: Since no delays are inserted into the fuzzy region, the proof that the frame size has been reduced in the fuzzy region is analogous to the proof of the previous lemma.

Before the fuzzy region, the situation is more complex. By the kth step, $0 \le k \le I^3$, a packet with delay x has waited xk/I^3 times. Thus, the delay of a packet at the kth step varies essentially uniformly from 0 to $u = k/I$. For $u \ge \log^3 I$, or equivalently, $k \ge I\log^3 I$, we can show that the relative congestion in any frame of size I_1 or greater has not increased much.

The proof uses the Lovasz local lemma as before. The calculation for the dependence is unchanged. The probability p_2 that more than $r_2 I_1$ packets use an edge e in a particular I_1-frame is given by

$$p_2 \le \sum_{s=r_3 I_1}^{r_1(I_1+u)} \binom{r_1(I_1+u)}{s} (I_1/u)^s (1-I_1/u)^{r_1(I_1+u)-s}.$$

Using the same inequalities as before, we have

$$p_2 \le O(r_1(I_1+u)e^{-r_1 I_1 \varepsilon_2^2(1/2-\varepsilon_2/2-I_1/\varepsilon_2^2 u-2I_1/\varepsilon_2 u)}).$$

For $I_1 = \log^2 I$, $u \ge \log^3 I$, it suffices that $\varepsilon_2 = O(1)/\sqrt{\log I}$. □

For steps 0 to $I\log^3 I$, we use the following lemma to bound the frame size and relative congestion.

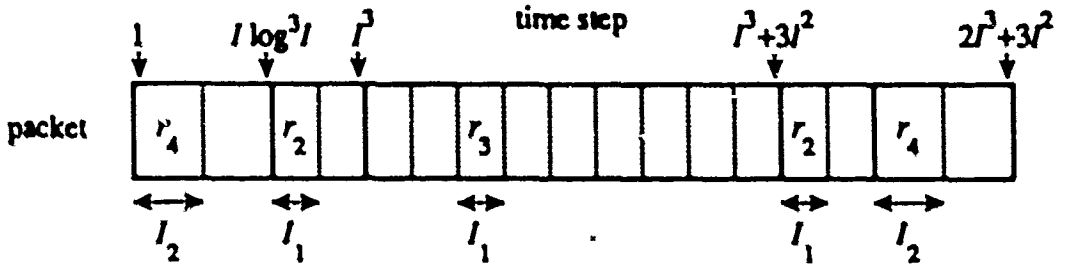

Figure 1-9: Final bounds on frame size and relative congestion. To reduce the frame size in the fuzzy regions, delays are inserted only outside the shaded region. Here $I_1 = \log^2 I$, $I_2 = \log^4 I$, $r_2 = r(1+O(1)/\sqrt{\log I})$, $r_3 = r(1+O(1)/\sqrt{\log I})$, and $r_4 = r_1(1+1/\log I) = r(1+O(1)/\sqrt{\log I})$.

Lemma 8 *The relative congestion in any frame of size I_2 or greater between steps 0 and $I\log^3 I$ is at most r_4, where $I_2 = \log^4 I$ and $r_4 = r_1(1+1/\log I)$.*

Proof: The proof is similar to that of Lemma 4. □

We have now completed our transformation of schedule S_i into schedule S_{i+1}. Let us review the relative congestion and frame sizes in the different parts of a block. Between steps 0 and $I\log^3 I$, the relative congestion in any frame of size I_2 or greater is at most r_4. Between this region and the fuzzy region, the relative congestion in any frame of size I_1 or greater is at most r_2. In the fuzzy region, the relative congestion in any frame of size I_1 or greater is at most r_3. After the fuzzy region, the relative congestion in any frame of size I_1 or greater is again r_2, until step $2I^3+3I^2-I\log^3 I$, where the relative congestion in any frame of size I_2 or greater is r_4. These bounds are shown in Figure 1-9. For the entire block it is safe to say that the relative congestion in any frame of size $I^{(i+1)} = \log^4 I$ or greater is at most $r^{(i+1)} = r(1+O(1)/\sqrt{\log I})$. □

1.3 On-line algorithms

1.3.1 An $O(c+d\log N)$ on-line algorithm

By applying the type of probabilistic analysis used in Section 1.2, it is fairly straightforward to schedule any set of N packets in $O(c+d\log N)$ steps with queues of size $O(\log N)$. We

simply delay the start of each packet by a random amount that is chosen uniformly from $[1, \frac{c}{\log N}]$, and then route all the packets forward in a synchronized fashion. More precisely, we introduce the initial delays and then consider the unconstrained schedule without regard for the rule that at most one packet traverse any edge in a single step. With high probability, no more than $O(\log N)$ packets will want to traverse any edge at any step of the unconstrained schedule. Hence we can simulate each step of the unconstrained schedule with $O(\log N)$ steps of a legitimate schedule. The final schedule consumes $O((d + \frac{c}{\log N})\log N) = O(c + d\log N)$ steps to complete the routing and uses queues of size $O(\log N)$.

1.3.2 An $O(c + d + \log N)$ on-line algorithm for layered networks

In this section we show how to route N packets whose paths have congestion c on a bounded-degree layered network with levels 0 through d in $O(c + d + \log N)$ steps with high probability using constant-size queues. A packet can originate at any node in the network, but its destination must be on a level with a larger number. No bound is placed on the size of the initial and final queues. The edge queues, however, can each hold at most q packets. The value of q can be any constant integer (including 1), and will affect the overall routing time by a constant factor. Each node has has in-degree and out-degree at most Δ, where Δ is a fixed constant.

The scheduling algorithm is identical to Ranade's algorithm except that instead of ordering the packets based on destination address, we order them according to random ranks. In particular, each packet is assigned a random rank chosen randomly, independently, and uniformly from the range $[1, w]$, where w will be specified later. A packet is routed through a node only after all the other packets with lower ranks that must pass through the node have done so. Ties in rank are broken according to destination address.

The routing protocol guarantees that the packets in each queue are arranged from head to tail in order of increasing rank. Before routing begins, the packets in each initial queue are sorted according to rank. At the tail of each initial queue there is a special end-of-stream (EOS) packet with the largest possible rank. All queues operate in a first-in first-out (FIFO) manner. At each step, a node examines the heads of its initial and input edge queues. If any of these queues are empty, then the node does nothing. Otherwise, it selects the packet with the smallest rank as a candidate to be transmitted. The candidate is sent forward only if the edge

queue that it must enter contains fewer than q packets at the beginning of the step. Thus, an edge queue is guaranteed never to hold more than q packets.

To prevent queues from becoming empty, whenever a node transmits a packet along one output edge, it sends a *ghost packet* with the same rank along all of its other output edges. The rank of the ghost packet provides the node on the next level with a lower bound on the ranks of the packets that it will receive in the future. Ghost packets allow a node to transmit a packet without having to wait for actual packets (if any) of higher rank to arrive on all of its input edges. Thus, a node starts transmitting packets as soon as it has received some kind of packet on each of its input edges, and at each step thereafter, it transmits a packet on all of its output edges until it sends an EOS packet. For simplicity we will assume that the queue size is at least two, so that once a queue contains a packet, it does not become empty until the node transmits an EOS packet. With minor modifications, the analysis can be made to work with queues of size one.

A ghost never remains at a node for more than one step and never resides in a queue except at the head. At the end of each step, a node first destroys any ghosts that were present in its edge queues at the beginning of the step, then destroys any ghosts not at the head of a queue.

To prove that the algorithm completes the routing in $O(c + d + \log N)$ steps, we use the same delay path argument as Ranade [81] (which, in turn is quite similar to the ones used by Aleliunas [3] and Upfal [94]), but we simplify the counting part of the analysis. The simplified counting has the additional nice feature that it allows the edge queue size to be as small as one, which was not possible with Ranade's original analysis.

A delay sequence has four components. The first is a path of length l that begins on level d at the destination of some packet. The path may traverse edges in either the forward direction (i.e., from a level i to a level $i+1$) or in the backward direction. If f is the number of forward edges traversed on the path, then $l \leq d+2f$. The second component is a sequence $s_1, \ldots, s_w$ of not necessarily distinct nodes on the path. The third component is a sequence $p_1, \ldots, p_w$ of w distinct packets such that the path for packet p_i passes through node s_i. The final component is a sequence $r_1, \ldots, r_w$ of ranks such that $r_i \leq r_{i+1}$.

Each delay sequence corresponds to a bad event in a probability space. The only use of randomness in the algorithm is in the choice of ranks for packets. Thus, the probability

space consists of w^N equally likely elementary outcomes, one for each possible setting of the ranks. A delay sequence corresponds to the event that the rank chosen for packet p_i is r_i, for $1 \leq i \leq w$. Each bad event consists of w^{N-w} elementary outcomes and occurs with probability $1/w^w$.

The following lemma is the crux of Ranade's argument.

Lemma 9 (Ranade) *For any w, if some packet is not delivered by step $d+w$ then a bad event corresponding to a delay sequence with $qf \leq w$ occurs.*

Corollary 10 *If no bad event occurs, then all of the packets are delivered within $d+w$ steps.*

The theorem below presents our simplified counting argument.

Theorem 11 *For any k_1, there is a k_2 such that the probability that any packet is not delivered by step $d+w$, where $w = k_2(d+c+\log N)$, is at most $1/N^{k_1}$.*

Proof: To bound the probability that some packet is delayed w steps, we need only bound the probability that some bad event occurs. This probability is at most

$$\frac{N(2\Delta)^l \binom{l+w}{w} (\Delta c)^w \binom{2w}{w}}{w^w}.$$

The numerator is an upper bound on the number of different delay sequences, each corresponding to a bad event. There are at most N places that the path can start, at most $(2\Delta)^l$ ways that it can continue, at most $\binom{l+w}{w}$ ways of selecting the nodes $s_1, \ldots, s_w$ on the path, at most $(\Delta c)^w$ ways to pick the packets $p_1, \ldots, p_w$ that pass through $s_1, \ldots, s_w$, and at most $\binom{2w}{w}$ ways to choose the ranks $r_1, \ldots, r_w$. Since the ranks are chosen from $[1, w]$, the probability that a bad event occurs is $1/w^w$. Using the inequality $l \leq d + 2f \leq d + 2w/q$, we see that for $w = \Omega(d + c + \log N)$, this probability can be made arbitrarily small, even if $q = 2$. □

For simplicity, we have heretofore ignored the possibility of *combining* multiple packets with the same destination. In many routing applications, there is a simple rule that allows two packets with the same destination to be combined to form a single packet, should they meet at a node. For example, one of the packets may be discarded, or the data carried by the two packets may be added to together. Combining is used in the emulation of concurrent-read concurrent-write parallel random-access machines [81] and distributed random-access machines.

If the congestion is to remain a lower bound when combining is allowed, then its definition must be modified slightly. The congestion of an edge is the number of different destinations for which at least one packet's path uses the edge. Thus, several packets with the same destination contribute at most one to the congestion of an edge.

If packets with the same destination are to be efficiently combined by the algorithm, then they must be given the same rank. For this purpose, a random hash function is used to generate ranks based on destination. Since ties in rank are broken according to destination, a node won't send a packet in one of its input queues unless it is sure that no other packet for the same destination will arrive later in the other queue. Thus, at most one packet for each destination traverses an edge.

For the counting argument to work, the ranks assigned by the hash function to any set of w packets must be independent. The universal hash function [17]

$$\text{rank}(x) = \left(\left(\sum_{i=0}^{w-1} a_i x^i\right) \bmod P\right) \bmod w \quad .$$

maps a destination $x \in [0, P-1]$ to a rank in $[0, w-1]$ with w-way independence. Here P is a prime number and the coefficients $a_i \in Z_P$ are chosen at random. The random coefficients use $O(w \log P)$ random bits. In fact, it suffices to choose ranks in the range $[0, \log N - 1]$ such that any set of $\log N$ are independent [63, 82]. In most applications, the number of possible different destinations is at most polynomial in N, so the hash function requires only $O(\log^2 N)$ bits of randomness.

1.3.3 Applications

In Sections 1.4 through 1.9 we examine the many applications of the $O(c + d + \log N)$-step scheduling algorithm for layered networks. These applications include routing algorithms for meshes, butterflies, multidimensional arrays and hypercubes, the shuffle-exchange graph, and fat-trees. Section 1.4 presents the simplest application: routing N packets in $O(\sqrt{N})$ steps on a $\sqrt{N} \times \sqrt{N}$ mesh. Another simple application, described in Section 1.5, is an algorithm for routing N packets in $O(\log N)$ steps on an N-node butterfly. The mesh and butterfly results were previously known [82, 81], but are included for completeness. Next, Section 1.6 presents an algorithm for routing kN packets on an N-node k-dimensional array with maximum side

length M in $O(kM)$ steps.

It is not obvious that the scheduling algorithm can be applied to the shuffle-exchange graph because it is not layered. Nevertheless, in Section 1.7 we show how to route N-packets in $O(\log N)$ steps on an N-node shuffle-exchange graph by identifying a layered structure in a large portion of the graph. In Section 1.8, we show how to adapt the scheduling algorithm to route a set of messages with load factor λ in $O(\lambda + \log M)$ steps on fat-tree [56] with root capacity M. The fat-tree routing algorithm leads to the construction of an N-node network with area $\Theta(N)$ that can simulate any other network of area $O(N)$ with slowdown $O(\log N)$. Finally, in Section 1.9 the scheduling algorithm is used as a subroutine in an $O(\log N)$-step sorting algorithm for the butterfly.

1.4 Routing on meshes

A 5×5 *mesh* is illustrated in Figure 1-10. Each node has a distinct label (x, y), where x is its column and y is its row. In an $n \times n$ mesh, $0 \leq x, y \leq n-1$. Thus, an $n \times n$ mesh has $N = n^2$ nodes. For $x < n-1$, node (x,y) is connected to $(x+1, y)$, and for $y < n-1$, node (x,y) is connected to $(x, y+1)$. Sometimes *wraparound* edges are included, so that a node labeled $(x, n-1)$ is connected to $(x, 0)$ and a node labeled $(n-1, y)$ is connected to $(0, y)$.

It is straightforward to apply the algorithm described in Section 1.3 to route N packets on a $\sqrt{N} \times \sqrt{N}$ mesh in $O(\sqrt{N})$ steps. The algorithm consists of four phases. In the first phase only those packets that need to route up and to the right are sent. The paths of the packets are selected greedily with each packet first traveling to the correct row, and then to the correct column. The level of a packet is the sum of its row and column numbers. This simple strategy guarantees that both the congestion and dilation of the phase are $O(\sqrt{N})$. The up-right phase is followed by up-left, down-right, and down-left phases. This algorithm was first discovered by Ranade [82]. Although $O(\sqrt{N})$-step routing algorithms for the mesh were known before [41, 43, 97], they all have more complicated path selection strategies.

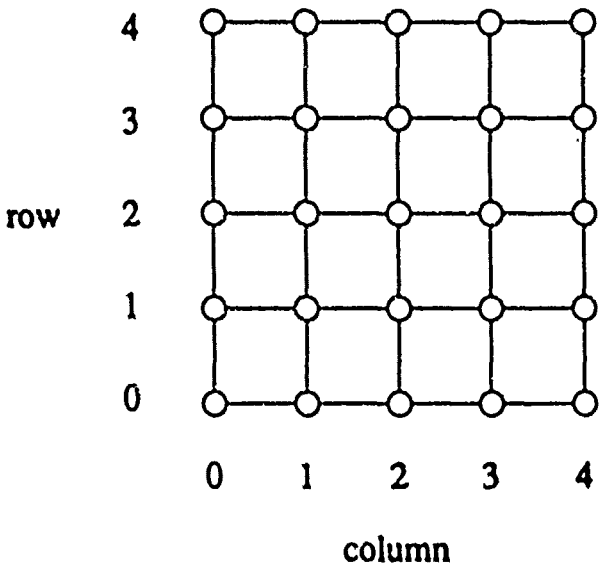

Figure 1-10: A 5×5 mesh.

1.5 Routing on butterflies

An 8-input *butterfly network* is illustrated in Figure 1-11. Each node has a distinct label $\langle l, r \rangle$, where l is its level, and r is its row. In an n-input butterfly, the level is an integer between 0 and $\lg n$, and the row is a $\lg n$-bit binary number. The nodes on level 0 and $\lg n$ are called the inputs and outputs, respectively. Thus, an n-input butterfly has $N = n(\lg n + 1)$ nodes. For $l < \lg n$, a node labeled $\langle l, r \rangle$ is connected to nodes $\langle l+1, r \rangle$ and $\langle l+1, r^{(l)} \rangle$, where $r^{(l)}$ denotes r with the lth bit complemented. Sometimes the input and output nodes in each row are identified as the same node. In this case the number of nodes is $N = n \lg n$. The butterfly has several natural recursive decompositions. For example, removing the nodes on level 0 (or $\lg n$) and their incident edges leaves two $n/2$-input subbuttterflies.

An important related network called the Benes network [10] is shown in Figure 1-12. An n-input Benes network has $2 \log n + 1$ levels and contains 2 n-input butterflies as edge-disjoint subgraphs. The two butterflies share nodes only on level $\log n$. The first butterfly has its inputs on level 0 of the Benes network, and its outputs on level $\log n$. The second is the mirror image of the first. It has its inputs on level $2 \log n + 1$, and its outputs on level $\log n$. An n-input

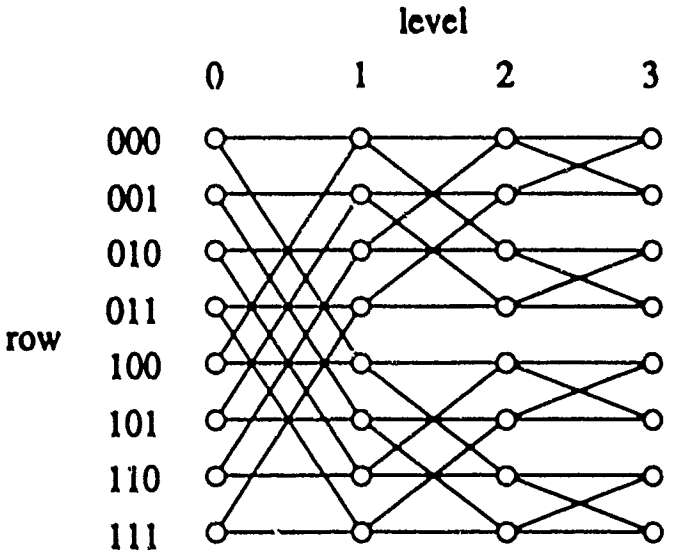

Figure 1-11: An 8-input butterfly network. Each node has a level number between 0 and 3, and a 3-bit row number. A node on level l in row r is connected to the nodes on level $l+1$ in rows r and $r^{(l)}$, where where $r^{(l)}$ denotes r with the lth bit complemented.

butterfly can emulate an n-input Benes network with constant slowdown. Waksman [98] proved that the inputs and outputs of a Benes network can be connected in any permutation by a set of node-disjoint paths.

Ranade [81] showed that the scheduling algorithm for layered networks can be applied to an N-node butterfly to route N packets in $O(\log N)$-steps using constant size queues. Routing is performed on a logical network consisting of $4 \lg n + 1$ levels. The first $\lg n$ levels of the logical network are linear arrays. The packets originate in these arrays, one to a node. Levels $\lg n$ through $3 \lg n$ form a Benes network. The last $\lg n$ levels are again linear arrays. Each packet has its destination in one of these arrays. Packets with the same destination are combined. The butterfly simulates each step of this network in a constant number of steps. Paths for the packets are selected using Valiant's paradigm; each packet travels to a random intermediate destination on level $2 \lg n$ before moving on to its final destination. This strategy ensures that with high probability the congestion is $O(\log N)$, so that the total time is $O(\log N)$.

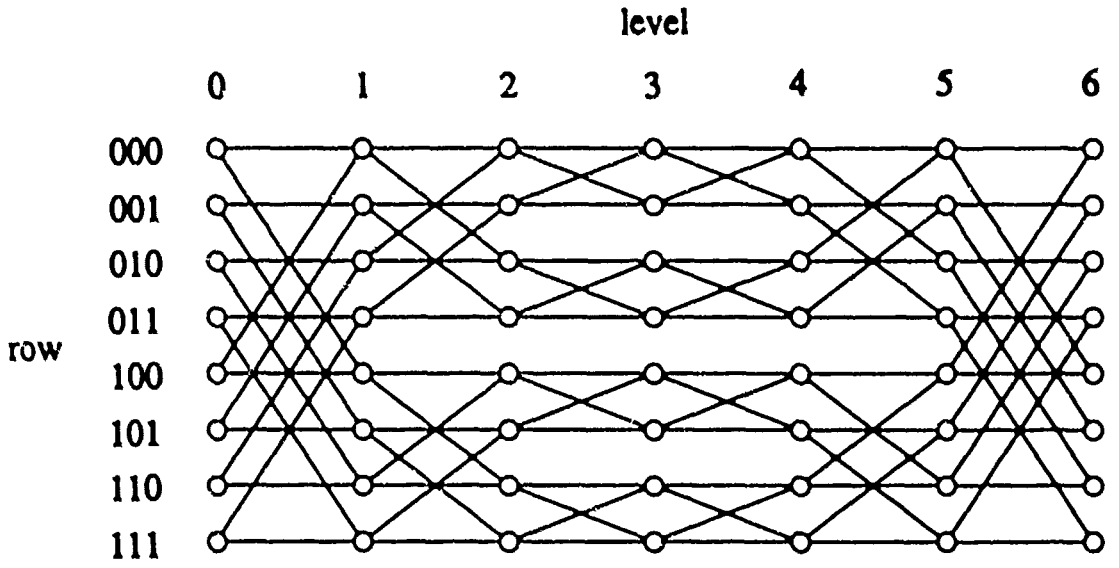

Figure 1-12: An 8-input Benes network consists of two back-to-back 8-input butterfly networks.

1.6 Routing on multidimensional arrays

In this section we describe a randomized algorithm for routing kN packets on an N-node k-dimensional array in $O(kM)$ steps using constant-size queues, where M is the maximum of the side lengths $M_1, \ldots, M_k$. Special cases include the mesh ($k = 2$) and the hypercube ($M = 2$). For arrays of dimension greater than two, no asymptotically-optimal constant-queue-size routing algorithms were previously known.

A k-dimensional array with side lengths $M_i \geq 2$, for $1 \leq i \leq k$, has $N = M_1 \cdots M_k$ nodes and kN edges. Each node has a distinct label $(w_1, \ldots, w_k)$, where $0 \leq w_i \leq M_i - 1$, for $1 \leq i \leq k$. A node has one outgoing and one incoming edge for each dimension; for $1 \leq i \leq k$, $(w_1, \ldots, w_k)$ has an edge to $(w_1, \ldots, w_i + 1 \bmod M_i, \ldots, w_k)$. We assume that at each step, a node may simultaneously transmit a packet on each of its k outgoing edges, and receive a packet on each of its k incoming edges.

In order to apply the scheduling algorithm from Section 1.3, routing is performed on a bounded-degree layered *logical network* that the array emulates. The logical network consists of $(2k+1)$ *plateaus* labeled 0 through $2k$, each consisting of N logical nodes. Each node in a plateau has a label $(w_1, \ldots, w_k)$ distinct from the labels of the other nodes in the plateau. We begin by

describing the edges in plateaus 0 through k. A node on plateau i has edges only in dimensions i and $i+1$. If $i > 0$ and $w_i < M_i - 1$, then the node labeled $(w_1, \ldots, w_k)$ has an edge to the node in the same plateau with label $(w_1, \ldots, w_i + 1, \ldots, w_k)$. Also, if $i < k$ and $w_{i+1} < M_{i+1} - 1$, then the node has an edge to $(w_1, \ldots, w_{i+1} + 1, \ldots, w_k)$. The only connections to plateau $i+1$ come from nodes with $w_{i+1} = M - 1$. For $i < k$, $(w_1, \ldots, w_i, M_{i+1} - 1, w_{i+2}, \ldots, w_k)$ on plateau i is connected to $(w_1, \ldots, w_i, 0, w_{i+2}, \ldots, w_k)$ on plateau $i+1$. Plateau k is connected to plateau $k+1$ by dimension 1 edges. Plateaus $k+1$ through $2k$ are essentially a copy of plateaus 1 through k. The edges on plateau $k+i$, $1 \leq i \leq k$ are given by the same rules as the edge on on plateau i. The level of node $(w_1, \ldots, w_k)$ in plateau i, $0 \leq i \leq k$, is $\sum_{j=1}^{k} w_j + \sum_{j=1}^{i} M_j$. For $k \leq i \leq 2k$, the level is $\sum_{j=1}^{k} w_j + \sum_{j=1}^{i-k} M_j + \sum_{j=1}^{k} M_j$. The network is layered because each edge connects a pair of nodes on adjacent levels.

Each step of the logical network can be emulated by the array in a constant number of steps. The array node labeled $(w_1, \ldots, w_k)$ emulates all of the logical nodes with the same label, one for each of the $2k+1$ plateaus. The array edge from $(w_1, \ldots, w_i, \ldots, w_k)$ to $(w_1, \ldots, w_i + 1 \bmod M_i, \ldots, w_k)$ emulates at most four logical edges, one each on plateaus $i-1$, i, $k+i-1$ and $k+i$.

Paths for the packets are selected using Valiant's paradigm. Initially each node on plateau 0 holds k packets in an initial queue. A packet travels from its origin on plateau 0 to a random destination on plateau k, then continues on to its true destination on plateau $2k$. Suppose that a packet originating at $(x_1, \ldots, x_k)$ on plateau 0 is to pass through $(r_1, \ldots, r_k)$ on plateau k on its way to $(y_1, \ldots, y_k)$ on plateau $2k$. In the first half of the path plateau i is used to make the ith component of the packet's location match the ith component of its random destination. The packet enters plateau $i \geq 1$ at node $(r_1, \ldots, r_{i-1}, 0, x_{i+1}, \ldots, x_k)$ and traverses dimension i edges to $(r_1, \ldots, r_i, x_{i+1}, \ldots, x_k)$. The packet then traverses dimension $i+1$ edges to $(r_1, \ldots, r_i, M_{i+1} - 1, x_{i+2}, \ldots, x_k)$ and crosses over to node $(r_1, \ldots, r_i, 0, x_{i+2}, \ldots, x_k)$ on plateau $i+1$. In the second half of the path, plateau $k+i$ is used to make the ith component of the packet's location match the ith component of the true destination in a similar fashion. The following lemma shows that with high probability, the congestion of the paths is at most $O(kM)$.

Lemma 12 *For any k_1, there is a k_2 such that the probability that $c > k_2 kM$ is at most $1/N^{k_1}$.*

Proof: For simplicity we analyze congestion in the first half of the network only. The calculation for the second half is identical.

We begin by bounding the probability that a particular edge is congested. There are two parts to the calculation: counting the number of packets that can possibly use the edge, and bounding the probability that an individual packet actually does so. First, we count packets that can use the edge. Consider an edge on plateau i from $(w_1, \ldots, w_k)$ to $(w_1, \ldots, w_i + 1 \bmod M_i, \ldots, w_k)$. Since a packet does not use any dimension $i+1$ through k edges before it uses a dimension i edge, any packet that uses the edge must come from an origin whose last $k-i$ components x_{i+1} through x_k match w_{i+1} through w_k. There are at most $M_1 \cdots M_i$ such origins, each transmitting k packets. Next we bound the probability that each of these packets actually uses the edge. A packet uses the edge only if components r_1 through r_{i-1} of its random destination match w_1 through w_{i-1}. The probability that these components match is $1/M_1 \cdots M_{i-1}$.

Since the random destinations are chosen independently, the number of packets, S, that pass through the edge has a binomial distribution. The probability that more than $k_2 kM$ packets use an edge is at most

$$\Pr[S > k_2 kM] \leq \binom{kM_1 \cdots M_i}{k_2 kM} \left(\frac{1}{M_1 \cdots M_{i-1}}\right)^{k_2 kM}.$$

Using the inequalities $M_i \leq M$ for $1 \leq i \leq k$, and $\binom{a}{b} \leq \left(\frac{ae}{b}\right)^b$, we have $\Pr[S > k_2 kM] \leq \left(\frac{e}{k_2}\right)^{k_2 kM}$.

To bound the probability that any edge is congested, we simply sum the probabilities that each particular edge is congested, i.e.,

$$\Pr[c > k_2 kM] \leq 4kN \left(\frac{e}{k_2}\right)^{k_2 kM}.$$

For any k_1, there is a k_2 such that this probability is at most $1/N^{k_1}$. □

Theorem 13 *For any k_1, there is a k_2 such that the probability that any packet is not delivered by step $k_2 kM$ is at most $1/N^{k_1}$.*

Proof: With high probability, the scheduling algorithm from Section 1.3 delivers all packets in $O(c+d+\log N)$ steps. The number of levels is $O(kM)$, and by Lemma 12 with high probability the congestion is $O(kM)$. Also, $\log N \leq kM$. □

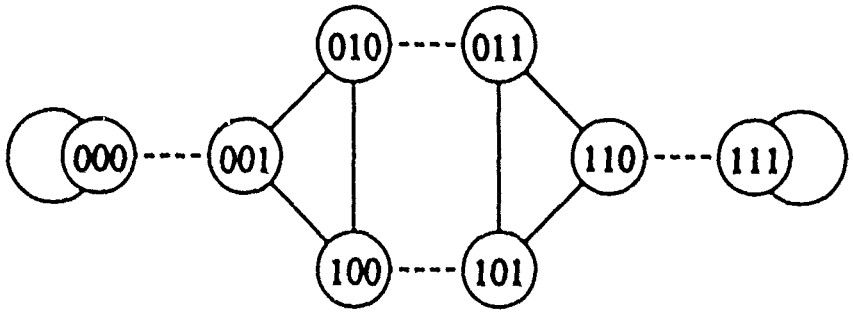

Figure 1-13: An 8-node shuffle-exchange graph. Shuffle edges are solid, exchange edges dashed.

1.7 Routing on shuffle-exchange graphs

In this section, we present a randomized algorithm for routing any permutation of N packets on an N-node shuffle-exchange graph in $O(\log N)$ steps using constant-size queues. The previous $O(\log N)$-time algorithms [3] required queues of size $\Omega(\log N)$.

Figure 1-13 shows an 8-node shuffle-exchange graph. Each node is labeled with a unique $\lg N$-bit binary string. A node labeled $a = a_{\lg N-1} \cdots a_0$ is linked to a node labeled $b = b_{\lg N-1} \cdots b_0$ by a *shuffle* edge if rotating a one position to the left or right yields b, i.e., if either $b = a_0 a_{\lg N-1} a_{\lg N-2} \cdots a_1$ or $b = a_{\lg N-2} a_{\lg N-3} \cdots a_0 a_{\lg N-1}$. Two nodes labeled a and b are linked by an *exchange* edge if a and b differ in only the least significant (rightmost) bit, i.e., $b = a_{\lg N-1} \cdots a_1 \overline{a}_0$. In the figure, the shuffle edges are solid, and the exchange edges are dashed.

The removal of the exchange edges partitions the graph into a set of connected components called *necklaces*. Each necklace is a ring of nodes connected by shuffle edges. If two nodes lie on the same necklace, then their labels are rotations of each other. Due to cyclic symmetry, the number of nodes in the necklaces differ. For example, in a 64-node shuffle-exchange graph, the nodes 010101 and 101010 form a 2-node necklace, while 011011, 110110, and 101101 form a 3-node necklace. For each necklace, the node with the lexicographically minimum label is chosen to be the necklace's *representative*.

1.7.1 Good and bad nodes

Unlike the mesh and butterfly networks, the shuffle-exchange graph cannot emulate a layered network in a transparent fashion. Nevertheless, it is still possible to apply the $O(c+d+\log N)$ scheduling algorithm for layered networks to the problem of routing on the shuffle-exchange graph. The key idea is that a large subset of the shuffle-exchange graph (at least $N/5$ nodes) can emulate a layered network. We call these nodes *good* nodes. The rest of the nodes are *bad*.

A node can be classified as bad for one of three reasons: (1) its label does not contain a substring of $\lg\lg N$ consecutive 0's (we consider the rightmost and leftmost bits in a label to be consecutive), (2) its label contains at least two disjoint longest substrings of at least $\lg\lg N$ consecutive 0's, or (3) its label is $0\cdots0$. Thus, the label of every good node contains a unique longest substring of 0's with length at least $\lg\lg N$. For simplicity, we assume that $\lg\lg N$ is integral, and that $\lg N \gg \lg\lg N$.

Since the length of a substring of consecutive 0's in a label is not changed by rotation, a necklace consists either entirely of good nodes or entirely of bad nodes. Furthermore, each good necklace consists of $\lg N$ good nodes since a unique longest substring of consecutive 0's precludes cyclic symmetry.

In order to route packets between all N nodes of the shuffle-exchange graph, we associate the bad nodes with good nodes. A type-1 bad node is associated with a good node by changing the least significant bit of its label to a 1 and the $\lg\lg N$ most significant bits to 0's. Each bad necklace of type 2 is associated with a good necklace by changing the two bits following the leading group of 0's in its representative's label to 01. Finally, the node $0\cdots0$ is associated with its neighbor $0\cdots01$.

Lemma 14 *At most* $4\lg N$ *bad nodes are associated with any good necklace.*

Proof: Each type-1 bad node is associated with the representative of a good necklace since, after the transformation, the longest string of consecutive 0's begins with the most significant bit. Only bad nodes whose labels differ from the representative's label in at most $\lg\lg N + 1$ bits are associated with it, so at most $2^{\lg\lg N+1} = 2\lg N$ type-1 bad nodes are associated with any good necklace.

To assess the number of type-2 bad nodes associated with a good necklace, we consider

the label of the representative of the good necklace and notice that only a bad necklace whose representative's label differs in the last bit of its leading block of 0's and possibly the bit after that can be mapped to the good necklace. Thus, at most two type-2 bad necklaces are associated with any good necklace.

Finally, no bad nodes of either type 1 or 2 are associated with the necklace of node $0\cdots01$. □

Corollary 15 *At least $N/5$ of the nodes are good.*

Proof: By Lemma 14 at most $4\lg N$ bad nodes are associated with any good necklace. Since every good necklace contains exactly $\lg N$ nodes, at least $N/5$ of the nodes are good. □

The remainder of this section provides the details of the routing algorithm. We begin by describing a logical layered network that the good nodes can easily emulate with constant overhead. Next, we show that, for any routing problem, choosing random intermediate destinations yields paths with congestion and dilation $O(\log N)$ in this network, with high probability. Thus, by applying the analysis of Section 1.3, routing on the logical network takes $O(\log N)$ steps, with high probability, and uses constant-sized queues. We conclude by describing the deterministic routing between good and bad nodes.

1.7.2 A layered network

The level of a node is determined by the distance to the representative node in its necklace. An alternate way to write a node's label is to place a line under its least significant bit (which we call the *current* bit), and then rotate it until it matches its representative's label. For example, 110001 can also be written 000$\underline{1}$11. The *level* of a node is the position of the current bit, counting from the left. For example, 000$\underline{1}$11 lies on level 3. (Note that the representative node lies on level $\lg N - 1$.)

The problem with this leveling scheme is that although it induces a leveling of the shift edges, it does not necessarily induce a leveling of the exchange edges. An exchange edge may create a new longest substring of 0's by appending two substrings separated by a single 1, and thus connect two levels which are very far apart.

To overcome this difficulty, we replace the exchange edges with *flip* edges. A flip edge links nodes labeled a and b if both are good, $a = a_{\lg N-1} \cdots a_j \cdots a_0$, $b = a_{\lg N-1} \cdots \overline{a}_j \cdots a_0$, and a_j is not in the longest block of 0's of a. Note that a flip edge extends a group of 0's by at most one. Thus no flip edge can create a new leading group of 0's, because if it grew a shorter group to be as big as the leading group, then it would lead to a bad node of type 2, a contradiction since flip edges occur only between good nodes by definition. Thus flip edges are leveled. The operation of the flip edges can be emulated by the shuffle-exchange graph with only a constant factor of slowdown; each flip edge is composed of an exchange edge, a shuffle edge, and possibly another exchange edge.

We denote by A the network composed of the good nodes, the shuffle edges (excluding the shuffle edges from level $\lg N - 1$ to 0), and the flip edges. Note that in network A, from any level 0 node we can reach any necklace with a longest string of 0's having the same or greater length by correcting bits starting from the end of the leading block of 0's.

In fact, we wish to be able to get from the level 0 node of necklace to *any* other necklace. Thus we append a mirror image A to itself so that we can reach necklaces with fewer 0's. The leveling is extended in the natural manner. We call this whole thing network AA^r, and note that network A can easily emulate it.

We denote by L the network consisting of the shuffle edges on the good nodes again excluding shuffle edges from level $\lg N - 1$ to level 0. Our method of path selection consists of routing from a good node to its level 0 node, then routing to a random intermediate necklace, then routing to the destination necklace, and finally routing to the appropriate good node. Thus, we route in a layered network composed of network L, network AA^r, another network AA^r, followed by network L. We extend the leveling in the natural manner and note that network A can easily emulate the whole thing.

1.7.3 Path selection and congestion

For each packet we choose its path by uniformly choosing a random good necklace to route through before going to its final destination. So the path for a packet consists of a path through L to node 0 of its necklace, the path through AA^r to its random intermediate necklace, the path through the second AA^r to its destination necklace, and a path through the second L

to the proper node of the necklace.

The following lemma shows that if at most $O(\log N)$ packets originate and terminate in each good necklace, then this method yields paths with congestion $O(\log N)$ with high probability.

Lemma 16 *Suppose that each good necklace sends and receives at most $b \lg N$ packets, where b is a fixed constant. Then for any constant k_1, there is a constant k_2 such that the probability that more than $k_2 \lg N$ packets use any edge is at most $1/N^{k_1}$.*

Proof: We observe that for the paths in the copies of L, we have congestion $b \lg N$, since at most $b \lg N$ packets start or end in any good necklace. By symmetry we claim that the analysis of the path portions in both copies of AA^r is the same. Finally we recall that in AA^r, we route packets going to necklaces with same or more 0's to the appropriate necklace in network A and straight across network A^r, and we route the other packets straight across in network A and use A^r to route to the proper necklace. We will show that any destination necklace gets $O(\log n)$ packets with high probability, so the straight across portion of the paths should not be a problem. To finish, we give the analysis of the congestion due to packets in just network A, and claim that the arguments will hold by symmetry for AA^r.

Consider an edge in the first copy of network A. In this half, packets going to necklaces with fewer leading 0's are routed straight across A. There are at most $b \lg N$ of these, so without loss of generality we ignore them. Suppose that e traverses levels m and $m+1$. Let x be the number of 0's in the necklace to which e goes. If $m < x$, then no packet from any other necklace uses e, since we only map to a necklace via flip edges after its longest string of 0's. Otherwise, we consider the number of packets from other necklaces that can use e. We know that only packets from at most 2^l other necklaces with $l = m - \lg\lg N$ could have used e since at most l bits could have changed by level $m+1$. Thus the number of packets that can use e is at most $b \cdot 2^l \lg N$ since each necklace starts with at most $b \lg N$ packets. The probability that a specific packet uses e, is the number of necklaces that can be reached using e, at most $2^{\lg N - \lg\lg N - l}$ (i.e., necklaces which match e's necklace in the first $l + \lg\lg N$ bits), divided by the total number of good necklaces, at least $N/5 \lg N$, which is just $5/2^l$.

The probability that more than $k_2 \lg N$ packets use e is at most

$$\binom{b \cdot 2^l \lg N}{k_2 \lg N} \left(\frac{5}{2^l}\right)^{k_2 \lg N},$$

since there are $b \cdot 2^l \lg N$ Bernoulli trials, each succeeding with probability $5/2^l$. The probability that any of the $O(N)$ edges of this stage has congestion more than $k_2 \lg N$ is $O(N)$ times this probablity. For any k_1, we can bound the product by $1/N^{k_1}$ by choosing k_2 large enough. □

Because the congestion and number of levels are $O(\log N)$, with high probability, the time to route the packets between the good nodes is also $O(\log N)$, with high probability, and the queue size is constant.

1.7.4 Packets from bad nodes

In this section we show how to deterministically route the packets from the bad nodes to their associated good necklaces.

Lemma 17 *Packets from bad nodes are routed to the associated good necklaces deterministically in $O(\log N)$ time using constant-size queues.*

Proof: Recall that we associate a bad node of type 1 with the necklace represented by a 1 in the least significant or current bit plus $\lg\lg N$ 0's in the most significant bits. We route these packets in the shuffle exchange graph by flipping the current bit to a 1 and flipping $\lg\lg N$ bits to the right to 0's. Thus we map a bad node to a good necklace at its level $\lg\lg N$ node.

For any necklace, we have a binary tree, the leaves of which are mapped to the necklace. Each level of the tree corresponds to one of the $\lg\lg N + 1$ bits that were flipped. Therefore, we can route packets from the binary tree leaves to the necklace, and distribute them along the necklace deterministically. This is easily done in $O(\log N)$ time with constant queues. The routing from the necklace to the tree is equally trivial. But, we need to ensure that traffic from the separate binary trees does not interfere too much. This is easy since any bad node is in at most two binary trees; in at most one as a leaf since any node is mapped to exactly one good node, and in at most one as an internal node since the number of 0's between the current node and the closest 1 to the left determines a unique level and the rest of the bits determine a unique tree.

To finish, we consider bad nodes of type 2. These are nodes without a unique longest string of 0's. Here we extend one of the groups of 0's by one 0, making sure not to join two groups of 0's by inserting a 1, mimicking the flip operation. For any good necklace whose representative

is $0^k1...$ only the necklaces represented by $0^{k-1}10...$ and $0^{k-1}11...$ can be mapped to it. Again, at most two bad necklaces are associated with any good necklace.

For each packet in such a bad necklace we route it through the node connecting it to the appropriate good necklace. We perform this movement by pipelining the packets through the edge which connects the two necklaces. We see that this mapping maps at most one packet from the bad necklace to a node in the good necklace. Since we are basically routing on linear arrays of length at most $2\lg N$, $2\lg N$ steps suffice to route the packets appropriately. Thus, $4\lg N$ steps are sufficient to route the packets from two bad necklaces.

This finishes the description of the maps to and from all the bad nodes except for node $0\cdots0$, which is adjacent to node $0\cdots01$. □

1.7.5 Summary

The main result of this section is summarized in the following theorem.

Theorem 18 *With high probability, an N-node shuffle-exchange graph can route any permutation of N packets in $O(\log N)$ steps using constant-size queues.*

Proof: There are three phases to the algorithm. First, packets originating at bad nodes are deterministically routed to the good nodes with which they are associated. By Lemma 17 this phase requires $O(\log N)$ steps. Next, packets are routed between the good nodes on the logical network. Since at most $4\lg N$ bad nodes are associated with each good necklace, with high probability the congestion of the paths on the logical network is $O(\log N)$, Lemma 16. Thus, this phase requires $O(\log N)$ steps, with high probability. The packets are routed in $O(\log N)$ steps using the scheduling algorithm from Section 1.3. Finally, packets destined for bad nodes are deterministically routed from the good nodes to bad. By an analysis similar to that of Lemma 17, this phase also requires $O(\log N)$ steps. □

1.8 Construction of area and volume-universal networks

In this section we construct a class of point-to-point networks that are *area-universal* in the sense that a network in the class with N processors has area $O(N)$ and can, with high probability,

simulate in $O(\log N)$ steps each message-step of any shared-bus network of area $O(N)$. The simulation is optimal because a point-to-point network may require $\Omega(\log N)$ steps to simulate one step of a shared-bus network. The networks are based on the fat-trees of Greenberg and Leiserson [29] and the simulation uses the message routing algorithm from Section 3.

In a fixed-connection network, processors communicate via wires. Each processor has a bounded number of read and write pins. In a point-to-point fixed-connection network, each wire connects one read pin with one write pin. In each message-step, the processor with the write pin may transmit a message of $O(\log N)$ bits to the processor with the read pin. In a shared-bus fixed-connection network, a wire may connect many read and write pins. Such a wire is called a bus. In each message-step, any processors wishing to send messages make them available on their write pins. Then the messages at the write pins of each wire are *combined* by some simple rule to form a single message Combining is assumed to require a single message-step, regardless of the number of messages combined or the rule used.

Leiserson was the first to display a class of fixed-connection networks that could efficiently simulate any other network of the same area or volume. In [56] he showed that a fat-tree of area $O(N)$ can simulate in $O(\log^3 N)$ bit-steps each bit-step of any point-to-point fixed-connection network of area $O(N)$. The simulation used an off-line routing algorithm for fat-trees. On-line routing algorithms were later developed by Greenberg and Leiserson [29] and Park [73]. None of these routing algorithms are capable of combining messages to the same destination. As a consequence, no scheme for simulating shared-bus networks was known until now. A network that can simulate in $O(1)$ steps each step of any shared-bus network area of equal area was presented in [69]. However, the connections in this network are not fixed, but instead processors communicate via reconfigurable busses.

A fat-tree network is shown in Figure 1-14. Its underlying structure is a complete 4-ary tree. Each edge in the 4-ary tree corresponds to a pair of oppositely directed groups of wires called *channels*. The channel directed from the leaves to the root is called an up channel; the other is called a down channel. The *capacity* of a channel c, $\text{cap}(c)$, is the number of wires in the channel. We call the tree "fat" because the capacities of the channels grow by a factor of 2 at every level. A fat-tree of height m has $M^2 = 2^{2m}$ leaves and $M = 2^m$ vertices at the root.

It will prove useful to label the switches at the top and bottom of each channel. Let the

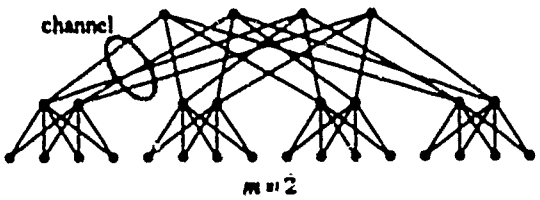

Figure 1-14: A fat-tree.

level of a switch be its distance from the leaves. Suppose a channel c connects $cap(c)/2 = 2^l$ switches at level l with $cap(c) = 2^{l+1}$ switches at level $l+1$. Give the switches at level l labels 0 through $2^l - 1$ and the switches at level $l+1$ labels 0 through $2^{l+1} - 1$. Then switch k at level l is connected to switches k and $k + 2^l$ at level $l+1$. The following lemma relates the labels of the switches on a message's path from a leaf to the root.

Lemma 19 *There is a unique shortest path from any leaf to a switch labeled k at the root, for $0 \leq k \leq M-1$, and that path passes through a switch labeled $k \bmod 2^l$ at level l, for $0 \leq l \leq m$.* □

For a set Q of messages to be delivered between the leaves of the fat-tree, we define the *load* of Q on a channel c, load(Q, c), to be the number of destinations of messages in Q for which at least one message must pass through c. Note that even if many messages with the same destination must pass through a channel, that destination contributes at most one to the load of the channel. We define the *load factor* of Q on c, $\lambda(Q, c)$, to be the ratio of the load of Q on c to the capacity of c, $\lambda(Q, c) = \text{load}(Q, c)/\text{cap}(c)$. The load factor on the entire network, $\lambda(Q)$ is simply the maximum load factor on any channel $\lambda(Q) = \max_c \lambda(Q, c)$. The load factor is a lower bound on the the number of steps required to deliver Q. We shall assume that $\lambda \leq M^k$, where k is some fixed constant. We shall sometimes write λ to denote $\lambda(Q)$ when the set of messages to be delivered is clear from the context.

In a *layered fat-tree* a switch at the top of an up channel at level l is connected to itself at the top of the corresponding down channel by a linear chain of switches of length $2(m-l)$. A message may only make a transition from an up channel to a down channel by traversing a chain. Thus all shortest paths between leaves in a layered fat-tree have length $2m$. Note that

the load of a set of messages on a channel of the layered fat-tree is identical to the load on the corresponding channel in the fat-tree.

The path that a message for destination x in column $2m$ takes through a layered fat-tree is determined by the m-universal hash function [17]

$$\text{path}(x) = \left(\left(\sum_{i=0}^{m-1} a_i x^i\right) \bmod P\right) \bmod M,$$

where P is a prime number larger than the number of possible different destinations, and the $a_i \in Z_P$ are chosen at random off-line. A message with destination x follows up channels until it can reach x without using any more up channels. It then crosses over to a down channel via a chain, and follows down channels to x. Note that a message only passes through a channel if it must. Also, all messages with destination x that pass through channel c pass through switch (path(x) mod cap(c)) at the top of c and through switch (path(x) mod (cap(c)/2)) at the bottom of c.

The following lemma shows that we can use the scheduling algorithm from Section 3 to route messages in a fat-tree.

Lemma 20 *For any constant c_1, there is a constant c_2 such that the probability that the number of steps required to deliver a set Q of N messages with load factor λ is more than $c_2(\lambda + \log M)$ is at most $1/M^{c_1}$, provided that N is polynomial in M.*

Proof: The paths of the messages are first randomized using the universal hash function path. With high probability, the resulting congestion is $c = O(\lambda + \log M)$. Each message travels a distance of $d = 2m = 2\log M$. The messages are then scheduled using the algorithm from Section 3. □

Let us now consider the VLSI area requirements [93] of fat-trees. A fat-tree with root capacity M and $\Theta(M^2)$ processors has a layout with area $O(M^2 \log^2 M)$ that is obtained by embedding the fat-tree in the tree of meshes[46]. The nodes of the tree of meshes in this layout are separated by a distance of $\lg M$ in both the horizontal and vertical directions. Thus, the $\Theta(\log M)$ space for the chain associated with each processor in the layered fat-tree can be allocated without increasing the asymptotic area of the layout. (In fact, it is possible to attach a chain of size $O(\log^2 M)$ to each fat-tree node without increasing the area by more than a

constant factor.) The leaves of the fat-tree are separated in the layout from each other by a distance of $\lg M$ in each direction. We can improve the density of processors without increasing the asymptotic area of the layout by connecting a $\lg M \times \lg M$ mesh of processors to each leaf. The resulting network has $\Theta(M^2 \log^2 M)$ processors and area $\Theta(M^2 \log^2 M)$. The N-processor network in this class has root capacity $\Theta(\sqrt{N}/\log N)$, $\Theta(N/\log^2 N)$ leaves, and area $\Theta(N)$.

The following theorem shows that this class of networks is area-universal.

Theorem 21 *With high probability, an N-processor point-to-point fixed-connection network U of area $\Theta(N)$ can simulate in $O(\log N)$ steps each step of any shared-bus fixed-connection network B of area $O(N)$.*

Proof: The processors of the shared-bus network B are mapped to the processors of the area-universal network U off-line using a recursive decomposition technique as in [56]. In each step, a wire of B is simulated by routing messages between the processors that it connects. At each level of the recursion at most $O(\text{cap}(c) \cdot \log N)$ wires connect the processors mapped below a channel c with the rest of the network. This property of the mapping ensures that the load factor of each set of messages used in the simulation of B is at most $O(\log N)$. At the bottom of the decomposition tree, a $O(\log N) \times O(\log N)$ region of the layout of B is mapped to each leaf of the fat-tree. The $O(\log N) \times O(\log N)$ mesh connected to the leaf in U simulates this region of B using standard mesh routing algorithms. □

The study of fat-tree routing algorithms that perform combining was motivated in part by an abstraction of the volume and area-universal networks called the distributed random-access machine (DRAM). A host of conservative algorithms for tree and graph problems for the exclusive-read exclusive-write (EREW) DRAM are presented in [58]. Recently we discovered conservative concurrent-read concurrent-write (CRCW) algorithms that require fewer steps for some of these problems. Until now, however, no efficient fat-tree routing algorithms that perform combining were known. The $O(\lambda + \log N)$ step routing algorithm presented here fills the void.

Only slight modifications to the area-universal fat-tree are necessary to make it volume universal[29]. The underlying structure of the volume-universal fat-tree is a complete 8-ary tree. Instead of doubling at each level, the channel capacities increase by a factor of 4. The

tree has m levels, root capacity $M = 2^{2m}$, and $M^{3/2} = 2^{3m}$ leaves. The switches at the top of a channel at level l are labeled 0 through $4^l - 1$. Switch k at level l is connected to switches k, $k + 4^l$, $k + 2 \cdot 4^l$, and $k + 3 \cdot 4^l$ at level $l + 1$. A layout with volume $O(M^{3/2} \log^{3/2} M)$ for the fat-tree can be obtained by embedding it in the three-dimensional tree of meshes. As before, a chain of size $O(\log^{3/2} M)$ can be attached to each node of the fat-tree without increasing the asymptotic layout area and the density of processors can be improved by connecting a $\lg^{1/2} M \times \lg^{1/2} M \times \lg^{1/2} M$ mesh to each leaf.

1.9 Sorting on butterflies

In this section we present a randomized algorithm for sorting $N \lg N$ packets on an $N \lg N$-node butterfly network in $O(\log N)$ steps using constant-size queues. The algorithm is based on the *Flashsort* algorithm of Reif and Valiant [84]. The main difference is that we use the algorithm for scheduling packets on layered networks in place of their scheduling algorithm, which requires queues of size $O(\log N)$. A similar approach has been suggested previously by Pippenger [76], and Reif [83].

1.9.1 The algorithm

The basic outline of the algorithm is the same as that of Flashsort. The first step is to randomly select a small set of *splitters* from among the packets that are to be sorted. Next the splitters are sorted deterministically. The splitters partition the packets into *intervals*. The ith interval consists of those packets whose keys are larger than the key of the $(i-1)$st largest splitter, and smaller than the key of the ith largest splitter. (We assume without loss of generality that all of the keys are distinct.) Using the splitters as guides, each interval of packets is routed to a different subbutterfly, where it is sorted recursively.

We begin by describing a recursive algorithm for sorting $N/\lg^{\alpha} N$ packets in $O(\log N)$ time on an $N \lg N$-node butterfly, where α is some fixed constant larger than one. The butterfly is "lightly loaded" by this factor of $\lg^{\alpha+1} N$ to ensure that, with high probability, at the lower levels of the recursion the number of packets to be sorted by each subbutterfly does not exceed the number of inputs to that subbutterfly. When the algorithm is invoked, each packet must

1. Count the number of packets entering the butterfly. Let the number of packets be denoted by n.
2. Randomly and independently, make each packet a candidate with probability $\sqrt{M}/n$.
3. Sort the candidates deterministically.
4. Select every $\lg N$th candidate to be a splitter.
5. Distribute the splitters for splitter-directed routing.
6. Route each packet to a random row of the butterfly.
7. Route each interval a subbutterfly via splitter-directed routing.
8. Distribute the packets in each interval to distinct inputs of the corresponding subbutterflies.
9. Sort the intervals recursively.

Figure 1-15: The steps performed by an M-input butterfly in the recursive algorithm for sorting $N/\lg^{\alpha} N$ packets in $O(\log N)$ time on an $N \lg N$-node butterfly using constant-size queues.

reside at a distinct input. As we shall see, this algorithm can be combined with Leighton's *Columnsort* algorithm [47] to sort all $N \lg N$ packets in $O(\log N)$ time.

The steps taken by a butterfly with M inputs are presented in some detail in Figure 1-15.

The first step in the algorithm is to count the number of packets entering the butterfly. Since the packets reside in distinct inputs, the total number of packets can be computed via a parallel prefix computation. The prefix computation can be performed in $O(\log M)$ time deterministically.

Next each packet independently chooses to be a splitter candidate with probability $\sqrt{M}/n$. As we shall see, with high probability the number of candidates is between $\sqrt{M}/2$ and $3\sqrt{M}/2$. This step requires only constant time.

The candidates are then sorted in $O(\log M)$ time using a simple deterministic algorithm based on counting [70, 84].

After the candidates are sorted, every $(\lg N)$th one in the sorted order is chosen to be a splitter. This *oversampling* technique, due to Reif, ensures that each of the intervals contains approximately the same number of splitters, with high probability. Note that we oversample

by a factor of $\lg N$, where N is the number of inputs in the entire network, independent of the number of inputs, M, of the butterfly on which the algorithm is invoked. Since with high probability there are at least $\sqrt{M}/2\lg N$ splitters, the subbutterflies at the next level of recursion should have at most $2\sqrt{M}\lg N$ inputs.

Next the splitters are distributed throughout the butterfly so that they can direct each interval of packets to the appropriate subbutterfly. We distribute a copy of the median splitter to each node in level 0 of the butterfly. Then we divide the splitters into upper and lower halves. We distribute a copy of the median splitter from the upper half to each node in the upper half of level 1. Similarly, we distribute a copy of the median splitter from the lower half to each node in the lower half of level 1. The process continues in this fashion until all of the splitters are used up. At this point, every node in the first $\Theta(\log(\sqrt{M}/\log N))$ levels of the butterfly has a copy of a splitter. This step can be performed deterministically in $O(\log M)$ time.

After the splitters are positioned, each packet is routed to a random row of the butterfly. The packets are scheduled using the algorithm for routing on layered networks.

Each interval of packets is then routed to a different subbutterfly. This step is called *splitter-directed routing* [84]. The paths of the packets are determined as follows. At level 0, each packet compares itself to the median splitter. If it is larger, it moves to the upper half of the second level, otherwise it moves to the lower half. The process is repeated at the level 1, with each packet being directed to the appropriate quarter of level 2, and so on. The packets are scheduled using the algorithm for routing on layered networks. When all the packets have been routed along in the butterfly as deeply as the splitters are assigned, each subbutterfly at that level picks new splitters and proceeds recursively.

The last step before the recursive call is to position the packets in each subbutterfly in distinct inputs. The problem of distributing a set of packets to distinct destinations is known as the *token distribution problem* [74]. On an M-input butterfly where at most c packets enter each input, M packets can be distributed deterministically in $O(c + \log M)$ time.

The recursion continues until either the number of inputs, M, is smaller than $2^{\sqrt{\lg N}}$, or the number of packets, n, is smaller than $\sqrt{M}$. In the first case, the sort is completed using Batcher's odd-even merge sort. An M-input butterfly can sort M packets in $O(\log^2 M)$ time using odd-even merge sort. For $M = 2^{\sqrt{\lg N}}$, the time is $O(\log N)$. In the second case, the

packets can be sorted deterministically in $O(\log M)$ time by the same technique that is used in step four to sort the candidates.

We can now make a rough estimate of the running time of this algorithm. Steps 1 and 2 are performed deterministically in $O(\log M)$ time. Assuming that there are $O(\sqrt{M})$ candidates, Steps 3, 4, and 5 also require $O(\log M)$ time. As we shall see, the expected time for Steps 6, 7 and 8 is $O(\log M)$. Although these steps sometimes take longer than expected, let us assume for now that they do not. In this case, the running time is given by the recurrence

$$T(M) \leq \begin{cases} T(2\sqrt{M}\lg N) + O(\log M) & M > 2^{\sqrt{\lg N}} \\ O(\log N) & M \leq 2^{\sqrt{\lg N}} \end{cases}$$

which has solution $T(N) = O(\log N)$.

1.9.2 Analysis

The analysis of the algorithm is broken into three parts, each corresponding to a different use of randomization in the algorithm. We first examine the use of randomization in selecting the splitters. We show that, with high probability, the number of splitters chosen by each butterfly is within a constant factor of the expectation and the number of packets in each interval is smaller than the number of inputs to the butterfly to which it is assigned. Next, we bound the probability that the congestion is large at any particular switch in Steps 6 and 7. Finally, we show that if the packets are scheduled using the randomized algorithm for layered networks, then it is unlikely that a delay of more than $O(\log N)$ will accumulate over the course of the algorithm.

1.9.3 Bounding the load

The first step in the analysis is to show that, with high probability, the number of splitter candidates chosen by each butterfly is within a constant factor of the expectation. We say that an M-input butterfly is *well-partitioned* if the number of splitter candidates chosen is between $\sqrt{M}/2$ and $3\sqrt{M}/2$. The $3\sqrt{M}/2$ upper bound ensures that the candidates can be sorted deterministically by the butterfly in $O(\log M)$ time and the $\sqrt{M}/2$ lower bound implies that the subbutterflies at the next level of recursion will have at most $2\sqrt{M}\lg N$ inputs. If all of the butterflies are well-partitioned, then the algorithm terminates after $O(\log\log N)$ levels of

recursion. (The choice of 1/2 and 3/2 as the coefficients of $\sqrt{M}$ are not particularly important. Other constants would serve equally well.)

Lemma 22 *For any fixed constant k_1 there is a constant k_2 such that the probability that any butterfly with at least $k_2 \lg^2 N$ inputs is not well-partitioned is at most $1/N^{k_1}$.*

Proof: We begin by considering a single M-input butterfly that is to sort n packets. Since each packet chooses independently to be a candidate, the number of candidates has a binomial distribution. Let S be the number of successes in r independent Bernoulli trials where each trial has probability p of success. Then we have $\Pr[S = s] = \binom{r}{s} p^s (1-p)^{r-s}$. We estimate the area under the tails of this binomial distribution using a Chernoff-type bound [18]. Following Angluin and Valiant [4] we have

$$\begin{aligned} \Pr[S \leq \gamma_1 rp] &\leq e^{-(1-\gamma_1)^2 rp/2} \\ \Pr[S \geq \gamma_2 rp] &\leq e^{-(1-\gamma_2)^2 rp/3} \end{aligned}$$

In our application $r = n$, $p = \sqrt{M}/n$, $\gamma_1 = 1/2$, and $\gamma_2 = 3/2$. For any fixed constant k_3, there is a constant k_2 such that the the right-hand sides of the two inequalities sum to at most $1/N^{k_3}$ for $M \geq k_2 \lg^2 N$.

To bound the probability that any butterfly is not well-partitioned, we sum the probabilities for all of the individual butterflies. Over the course of the algorithm, the algorithm is invoked on at most $N \lg N$ individual butterflies. Thus, the sum is at most $\lg N/N^{k_3-1}$. For any k_1, there is a k_3 such that this sum is at most $1/N^{k_1}$. □

The next lemma shows that, with high probability, the number of packets in each interval is at most a constant factor times its expectation. We say that an M-input butterfly that is assigned n packets to sort is *α-split* if every interval has size at most $\alpha n \lg N/\sqrt{M}$. As we shall see, if every butterfly is $O(1)$-split and there are $O(\log \log N)$ levels of recursion, then by lightly loading the butterfly we can ensure that no butterfly is assigned too many packets to sort.

Lemma 23 *For any fixed constant k_1 there is a constant k_2 such that the probability that every butterfly is k_2-split is at least $1 - 1/N^{k_1}$*

Proof: We begin by examining a single packet in a single M-input butterfly that is to sort n-packets. To show that a packet lies in an interval of size at most $k_2 n \lg N/\sqrt{M}$ it is sufficient

to show that both following and preceding it in the sorted order at least $\lg N$ of the next $k_2 n \lg N/2\sqrt{M}$ packets are candidates.

First we consider the packets that follow in the sorted order. The number of candidates in a sequence of $k_2 n \lg N/2\sqrt{M}$ packets has a binomial distribution. For $r = k_2 n \lg N/2\sqrt{M}$, $p = \sqrt{M}/n$, $rp = k_2 \lg N/2$, and $\gamma_1 = 2/k_2$, we have $\Pr[S \leq \lg N] \leq e^{-k_2(1-2/k_2)^2 \lg N/4}$. For any k_3 we can make the right-hand side smaller than $1/N^{k_3}$ by choosing k_2 large enough.

The calculations for the packets that precede in the sorted order are identical. The probability that fewer $\lg N$ of the preceding $k_2 n \lg N/2\sqrt{M}$ packets are candidates is at most $1/N^{k_3}$. Thus, the probability that an individual packet lies in an interval of size greater than $k_2 n \lg N/2\sqrt{M}$ is at most $2/N^{k_3}$.

To bound the probability that any interval in the butterfly is too large we sum the probabilities that each individual packet lies in an interval that is too large. Since there are at most $N \lg N$ packets, this sum is at most $2 \lg N/N^{k_3-1}$.

To bound the probability that any butterfly is not k_2-split, we sum the probabilities that each individual butterfly is not. Over the course of the algorithm, the algorithm is invoked on at most $N \lg N$ butterflies. The sum of the probabilities is at most $2\lg^2 N/N^{k_3-2}$. For any constant k_1, we can make this sum at most $1/N^{k_1}$ by making k_3 large enough. □

The remainder of the analysis is conditioned on the event that every butterfly is well-partitioned and $O(1)$-split, which occurs with high probability. Two technical points bear mentioning. First, Lemma 22 requires that the number of inputs to every butterfly be at least $k_2 \lg^2 N$, where k_2 is some constant. Since the recursion terminates when the number of inputs is $2^{\sqrt{\lg N}}$, N must be large enough that $2^{\sqrt{\lg N}} > k_2 \lg^2 N$. Second, both Lemmas 22 and 23 hold independent of the number of packets to be sorted by each butterfly. Thus, as the following lemmas show, we can adjust the load on the butterfly in order to ensure that each M-input butterfly receives at most M packets to sort.

Lemma 24 *The number of levels of recursion is* $O(\log \log N)$.

Proof: At each level of recursion the number of inputs drops from M to at most $2\sqrt{M}/\lg N$, until the number of inputs reaches $2^{\sqrt{\lg N}}$. □

Lemma 25 *There is an $\alpha > 0$ such that if the number of packets to be sorted is $N/\lg^{\alpha} N$, then the number of packets are assigned to any M-input butterfly is at most M.*

Proof: Since the ratio of packets to inputs is $1/\lg^{\alpha+1} N$ at the top level of the recursion, and increases by at most a constant factor at each of $O(\log\log N)$ levels, it is possible to choose α such that at the bottom level it will be at most one. □

1.9.4 Bounding the congestion at each switch

The second step in the analysis is to bound the probability that too many packets pass through any switch in Steps 6 and 7. The following lemma provides a bound on the probability that the congestion, c, in an M-input butterfly exceeds $\lg M$ in either of of these steps.

Lemma 26 *There is a fixed constant β_1 such that for $s > \lg M$,*

$$\Pr[c \geq s] \leq \left(\frac{\beta_1}{s}\right)^s.$$

Proof: For the sake of brevity, we examine Step 7 only. A similar (and simpler) analysis holds for Step 6.

We begin by counting the number of packets that can possibly use a switch. Let L denote the depth of an M-input butterfly, i.e., $L = \lg M$. From a switch at level l, $0 \leq l \leq L$, 2^{L-l} rows can be reached. The splitters partition these rows into subbutterflies. From the previous argument, the number of packets that enter each of these subbutterflies is at most the number of inputs, with high probability. Thus, at most 2^{L-l} packets can pass through the switch.

Next we determine the probability that a packet that can pass through the switch actually does so. A switch at level l can be reached from 2^l different inputs. Since each packet begins in a random input, the probability that it can reach the switch is 2^{l-L}.

The number of packets, S, that pass through a particular switch at level l has a binomial distribution. The number of trials is $r = 2^{L-l}$ and the probability of success is $p = 2^{l-L}$. Thus, $\Pr[S = s] = \binom{2^{L-l}}{s} \left(2^{l-L}\right)^s \left(1 - 2^{l-L}\right)^{2^{L-l}-s}$. Using the inequality $\binom{a}{b} \leq (ae/b)^b$, we have $\Pr[S = s] \leq (e/s)^s$. For $s \geq 1$, the right-hand side decreases by at least a constant factor with each increase of 1 in s. Thus $\Pr[S \geq s] \leq O\left((e/s)^s\right)$.

We bound the congestion in the entire butterfly by summing the individual probabilities over all $2^{O(L)}$ switches in the butterfly. We have

$$\Pr[c \geq s] \leq 2^{O(L)} \left(\frac{e}{s}\right)^s.$$

For $s \geq L$, we have $\Pr[c \geq s] \leq (\beta_1/s)^s$ for some constant β_1. □

1.9.5 Bounding the cumulative delay

Since a subbutterfly does not begin to execute its algorithm until the larger butterfly at the previous level of recursion is finished, delay in excess of the time allotted to each butterfly accumulates over the course of the algorithm. An M-input butterfly is allotted $O(\log M)$ time to perform its steps. However, Steps 6, 7, and 8 are not guaranteed to terminate in time $O(\log M)$. It is tempting to try to prove that these steps terminate quickly with high probability. This approach fails because at the lower levels of the recursion the problem size is so small that nothing can be ascertained with high probability. Instead we must argue that although delay may occur at any particular step, it is unlikely that a lot of delay will accumulate over a sequence of steps.

The delay from Step 8 is relatively easy to analyze. This step requires $O(c + L)$ time; the delay depends only on the congestion. Lemma 26 bounds the probability that the congestion is large.

There are two possible causes of delay in Steps 6 and 7. A poor set of random rows for the packets can cause congestion at some node, which guarantees that some packet will arrive at its destination late. On the other hand, even if the congestion is small, a poor choice for the random ranks used by the scheduling algorithm may delay a packet. The following pair of lemmas bounds the probability that the delay from these steps is large. The first is a restatement of the main scheduling theorem for layered networks. It bounds the probability that a packet will be delayed when the congestion is small. The second puts this bound together with the bound that the congestion is large from Lemma 26.

Lemma 27 *For a bounded-degree layered network with L levels and a set of $2^{O(L)}$ packets whose paths have congestion c, there is a fixed constant β_2 such that the probability that any packet arrives at its destination after time w, $w > L$, is at most $(\beta_2 c/w)^w$.*

Lemma 28 *There is a constant $\beta_3 > 1$ such that the probability Steps 6 and 7 require more than w time steps, $w > L$, is at most $2(1/\beta_3)^w$.*

Proof: For the sake of brevity, we examine Step 7 only. A similar analysis holds for Step 6.

We break the analysis into two cases according to whether the congestion is small or large. Let T be the time at which the last packet arrives. Then

$$\Pr[T \geq w] \leq \Pr[T \geq w | c < w/\beta_2\beta_3] + \Pr[c \geq w/\beta_2\beta_3].$$

We use Lemma 27 to bound the first term on the right. Plugging in $w/\beta_2\beta_3$ for c yields $\Pr[T \geq w | c < w/\beta_2\beta_3] \leq (1/\beta_3)^w$. We use Lemma 26 to bound the second term on the right. Plugging in $w/\beta_2\beta_3$ for c yields $\Pr[c \geq w/\beta_2\beta_3] \leq (\beta_1\beta_2\beta_3/w)^w$. Since $w > L \geq \sqrt{\lg N}$, and β_1, β_2, and β_3 are constants, $w > \beta_1\beta_2\beta_3^2$ for sufficiently large N. □

The following lemma bounds the combined delay of Steps 6, 7, 8.

Lemma 29 *There are constants β_4 and $\beta_5 > 1$ such that the probability that Steps 6, 7, 8 together require time $\beta_4 L + w$ is at most $(1/\beta_5)^w$.*

Proof: Step 8 can be performed deterministically in time $O(c + L)$. From Lemma 26 we have $\Pr[c \geq s] \leq (\beta_1/s)^s$, for $s > L$. For our purposes, a weaker bound on this probability suffices. Since β_1 is a constant, there is a constant k_1 such that $(1/k_1)^s \leq (\beta_1/s)^s$ for sufficiently large L. Combining this bound with that of Lemma 27 yields the desired result. □

To complete our analysis of the algorithm, we need to bound the probability that more than $O(\log N)$ delay accrues during the sort.

Lemma 30 *For any fixed constant k_1, there is a constant k_2 such that the probability that the cumulative delay is more than $k_2 \lg N$ is at most $1/N^{k_1}$.*

Proof: The cumulative delay at the bottom level of the recursion is the sum of the delay at each of the butterflies on the branch of the recursion tree from the top level to the leaf. Let D_i be the delay beyond $\beta_4 L$ at the ith level of the recursion. Then $\Pr[D_i = w] \leq (1/\beta_5)^w$. Notice that there is no dependence on i in this expression. Let D be the cumulative delay on a branch

of the recursion from the top level to a leaf. Then $D = \sum_{i=0}^{O(\log\log N)} D_i$. Generating functions help us here. The generating function for D_i is

$$G_{D_i}(z) = \sum_{w=0}^{\infty} \Pr[D_i = w] z^w,$$

where z^w can be thought of as a place holder. To sum the delay, we simply multiply the generating functions. Thus, the generating function for the cumulative delay is $G_D(z) = \prod_{i=0}^{O(\log\log N)} G_{D_i}(z)$. The coefficient of z^w in $G_D(z)$ is $\binom{w+O(\log\log N)}{w}(1/\beta_5)^w$. For $w = O(\log\log N)$, this coefficient is at most $(O(1)/\beta_5)^w$. For any k_3, there is a k_2 such that $\sum_{w=k_2 \lg N}^{\infty}(O(1)/\beta_5)^w$ is at most $1/N^{k_3}$.

To bound the probability that the cumulative delay exceeds $k_2 \lg N$ on any branch of the recursion, we sum the individual probabilities for all of the branches. There are at most N branches. Thus, the sum is at most $1/N^{k_3-1}$. For any k_1, there is a k_3 such that this sum is at most $1/N^{k_1}$. □

1.9.6 Putting it all together

Theorem 31 *With high probability, an $N \lg N$-node butterfly can sort $N \lg N$ packets in $O(\log N)$ steps using constant-size queues.*

Proof: The algorithm for sorting $N \lg N$ packets on an $N \lg N$-node butterfly uses the algorithm for sorting $N/\lg^\alpha N$ packets as a subroutine. First each packet independently chooses to be a splitter with probability $1/\lg^{\alpha+1} N$. With high probability, this leaves $\Theta(N/\log^\alpha N)$ candidates. The candidates are sorted using the subroutine. Then every $\lg N$th candidate is selected to be a splitter, leaving $\Theta(N/\log^{\alpha+1} N)$ splitters. The splitters are distributed throughout the butterfly, and splitter-directed routing is used to route intervals of size $O(\log^{\alpha+2} N)$ to subbutterflies with $\Theta(\log^{\alpha+1} N)$ inputs. Now each interval of $O(\log^{\alpha+2} N)$ packets resides in a group of $\Theta(\log^{\alpha+1} N)$ butterfly rows. Each of these rows contains $O(\log N)$ packets. The packets in each row can be sorted in $O(\log N)$ time using an odd-even transposition sort. With a fixed number of row sorts and permutations, all of the packets can be sorted in $O(\log N)$ time using Columnsort. □

1.10 Counterexamples to on-line algorithms

This section presents examples where several natural on-line scheduling strategies do poorly. Based on these examples, we suspect that finding an on-line algorithm that can schedule any set of paths in $O(c+d)$ steps using constant-size queues will be a challenging task.

In the first example, we describe an N-node network in which a set of packets with congestion and dilation $O(1)$ requires $\Omega(\log^2 N/\log\log N)$ steps to be delivered using the strategy of Section 1.3. This example does not contradict the results of Section 1.3, since the network has $\Theta(\log^2 N)$ levels. However, it shows that reducing the congestion and dilation below the number of levels will not necessarily improve the running time.

Observation 32 *For the strategy of Section 1.3, there is an N-node directed acyclic network of degree 3 and a set of paths with congestion $c = 3$ and dilation $d = 3$ where the expected length of the schedule is $\Omega(\log^2 N/\log\log N)$.*

Proof: The network consists of many disjoint copies of the subnetwork pictured in Figure 1-16. For simplicity, we dispense with the initial queues; the packets originate in edge queues. The subnetwork is composed of $k/\log k$ linear chains of length k, where k shall later be shown to be $\Theta(\log N)$. The second node of each linear chain is connected to the second to last node of the previous chain by a diagonal edge. We assume that at the end of each edge there is a queue that can store 2 packets. Initially, the queue into the first node of each chain contains an end-of-stream (EOS) signal and one packet, and the queue into the second node contains two packets. A packet's destination is the last node in the previous chain. Each packet takes the diagonal edge to the previous chain and then the last edge in the chain. Thus, the length of the longest path is $d = 3$.

When the ranks $r_1, \ldots, r_{3k/\log k}$ of the packets $p_1, \ldots, p_{3k/\log k}$ are chosen so that $r_i < r_{i+1}$ for $1 \leq i < 3k/\log k$, packet $p_{3k/\log k}$ requires $\Omega(k^2/\log k)$ steps to reach its destination. The scenario unfolds as follows. Packets p_1 and p_2 take a diagonal edge in the first two steps. These packets cannot advance until the EOS reaches the end of the first chain, in step k. In the meantime, ghosts with ranks r_1, r_2, and r_3, travel down the second chain, but packet p_3 blocks an EOS signal from traveling down the chain. Packets p_4 and p_5 are waiting for this EOS signal. They cannot advance until step $2k$. In this fashion, the delay is propagated down to packet

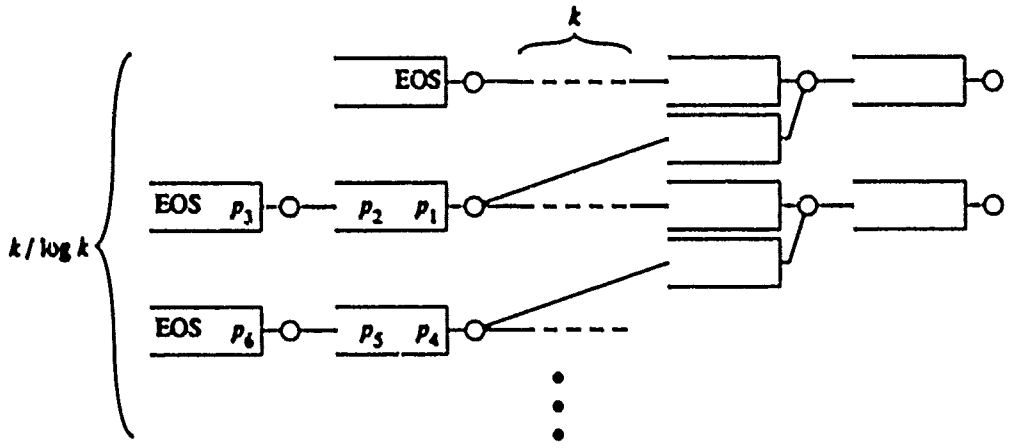

Figure 1-16: Example 1.

$p_{3k/\log k}$.

A simple calculation reveals that the probability that $r_i < r_{i+1}$ for $1 \leq i \leq 3k/\log k$ is $1/2^{\Theta(k)}$. Thus, if we have $2^{\Theta(k)}$ copies of the subnetwork, we expect the ranks of the packets to be sorted in one of them. For the total number of nodes in the network to be N, we need $k = \Theta(\log N)$. In this case, we expect some packet to be delayed $\Omega(\log^2 N/\log\log N)$ steps in one copy of the subnetwork. □

It is somewhat unfair to say that the optimal schedule for this example has length $O(c+d) = O(1)$, since ghosts and EOS signals must travel a distance of $\Theta(\log N)$. However, even if the EOS signals are replaced by packets with the appropriate ranks, the dilation is only $O(\log N)$, and thus the optimum schedule has length $O(\log N)$.

The second example is quite general. It shows that for any deterministic strategy that chooses the order in which packets pass through a switch independent of the future paths of the pac there is a network and a set of paths with congestion c and dilation d for which the schedule produced has length at least $c(d-1)/\log c$. This observation covers strategies such as giving priority to the packet that has spent the most (or least) time waiting in queues, and giving priority to the packet that arrives first at a switch. The network is a complete binary tree of height $d-1$ with an *auxiliary edge* from the root to an *auxiliary node.*

Observation 33 *For any deterministic strategy that chooses the order in which packets through*

a switch independent of the paths that the packets take after they pass through the switch, there is a network and a set of paths with congestion c and dilation d for which the schedule produced has length $c(d-1)/\log c$.

Proof: We construct the example for congestion c and dilation d, $E(c,d)$, recursively. The base case is the example $E(c, \log c + 1)$. Each of the c leaves sends a packet to the auxiliary node, causing congestion c in the auxiliary edge. The network for $E(c,d)$ contains c copies of the network for $E(c, d - \log c)$. First, the auxiliary nodes for theses copies are paired up and merged so that there are $c/2$ auxiliary nodes each with two auxiliary edges into it. Next, the auxiliary nodes become the leaves of a complete binary tree of height $\log c - 1$ with its own auxiliary node and edge. For each copy of $E(c, d - \log c)$, the deterministic scheduling strategy chooses some packet to cross its auxiliary edge last. We extend the path of this packet so that it traverses the auxiliary edge in $E(c,d)$. The dilation of the new set of paths is d and the congestion c. The length of the schedule, $T(c,d)$, is given by the recurrence

$$T(c,d) \geq \begin{cases} T(c, d - \log c) + \log c - 1 + c & d > \log c + 1 \\ \log c + c & d = \log c + 1 \end{cases}$$

and has solution $T(c,d) \geq c(d-1)/\log c$. □

The third example shows that the simple look-ahead strategy of giving priority to the packet with the farthest distance left to travel fails as well.

Observation 34 *For the strategy in which the packet with the farthest distance left to travel (or the farthest total distance to travel) is given priority, there is an N-node network with diameter* $O(\sqrt{N})$ *and a set of paths with congestion* $O(\sqrt{N})$ *and dilation* $O(\sqrt{N})$ *for which the schedule produced has length* $\Omega(N)$.

Proof: The network consists of k linear chains labeled 0 through $k-1$. Chain i is composed of $3k-2-i$ nodes labeled 0 through $3k-3-i$. It meets chain $i+1$ at node $k-1-i$ and at every second node thereafter up to node $k+i-1$. Figure 1-17 shows the network for $k=4$. We assume that the queue the end of each edge has unlimited size and that at each step a node can send at most one packet. Initially, the first node of each chain holds k packets. The destination of each of these packets is the end of the chain. Note that packets in chain i have

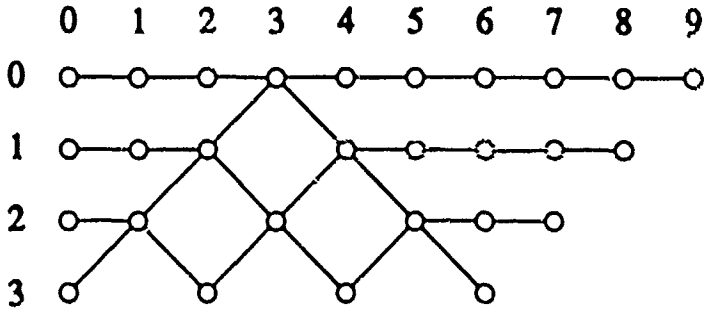

Figure 1-17: Example 3.

higher priority than those in the chain $i+1$ whenever they meet since the chain i packets must travel one step farther than those in chain $i+1$.

The key to this example is that the packets in chain $i+1$ are delayed by all of the packets in chain i at every meeting point between chains i and $i+1$, Since the packets in chain 0 are never delayed and the packets in chain 1 are not delayed by any packets other than those in chain 0, the packets in these two chains arrive at their one meeting point simultaneously. At this meeting point, the packets in chain 0 have priority and delay the packets in chain 1 by k steps. In general, the packets in chains i and $i+1$ arrive at meeting point j simultaneously because the packets in chain i have been delayed $j-1$ times by chain $i-1$ and the packets in chain $i+1$ have been delayed $j-1$ times by the chain i.

The claim implies the theorem for $k=\sqrt{N}$. The packets in chain $k-1$ are delayed by k packets at each of $k-1$ meeting points, resulting in a total delay of $\Omega(N)$. □

The fourth example shows that the natural strategy of assigning priorities to the packets at random is not effective either.

Observation 35 *For the strategy of assigning each packet a random rank and giving priority to the packet with the lowest rank, there is an N-node network with diameter $O(\log N/\log\log N)$ and a set of paths with dilation $d=O(\log N/\log\log N)$ and congestion $c=O(\log N/\log\log N)$ where the expected length of the schedule is $\Omega((\log N/\log\log N)^{3/2})$.*

Proof: As in Example 1, the network consists of many copies of a subnetwork. Each subnetwork is constructed so that $d=c=k/\log k$. A subnetwork consists of a linear chain of length d,

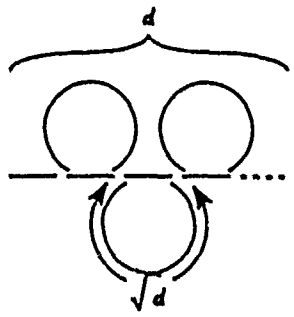

Figure 1-18: Example 4.

with loops of length $\sqrt{d}$ between adjacent nodes (see Figure 1-18). The packets are broken into $\sqrt{d}$ groups numbered 0 through $\sqrt{d}-1$ of $\sqrt{d}$ packets each. The packets in group i use the linear chain for $i\sqrt{d}$ steps and then use $\sqrt{d}-i$ loops as their path. As in Example 3, we assume that queues have unlimited capacity and that at each step a node can send a single packet.

If the random ranks are assigned so that the packets in group i have smaller ranks than the packets in groups with larger numbers, then the packets in group i delay the packets in groups with larger numbers by $d - i\sqrt{d}$ steps. Thus the last packet experiences an $\Omega(d\sqrt{d}) = O((k/\log k)^{3/2})$ delay.

Once again the ranks of the packets must have a specific order, which can be shown to happen with high probability given enough copies of the subnetwork. As in Observation 32, it is not hard to show this requires $k = \Theta(\log N)$. □

1.11 Remarks

The scheduling algorithm from Section 1.3 can be used as a subroutine in algorithms for emulating shared-memory machines on bounded-degree networks. A shared-memory machine with a large address space can be emulated by randomly hashing the memory locations to the nodes of a butterfly as in [35] and [81]. The hashing ensures that the congestion of the packets implementing each memory access step is small. The algorithm from Section 1.3 can be used to

schedule the the movements of these packets.

The algorithm for sorting on the butterfly with constant-size queues can modified to sort kM^k packets on a k-dimensional mesh with side length M in $O(kM)$ time using constant-size queues.

Given a set of n packets whose paths have congestion c on a layered network with d levels, a setting of ranks that ensures delivery in time $O(c + d + \log n)$ can be found can be found off-line deterministically in time $2^{O(c+d+\log n)}$. The proof uses the Raghavan-Spencer technique [78, 89] to sequentially find a setting of the ranks so that no bad event corresponding to a delay sequence occurs.

One application is in preparing simulations by volume and area-universal networks off-line so that no random bits are needed. As before, the first step is to map the processors of the network to be simulated, B, to the processors of the area-universal network, U, from Section 1.8 using the recursive decomposition strategy from [56]. Network U has N processors, and B has area $O(N)$. To simulate each step of B, network U must route a set of $n = O(N/\log N)$ messages with load factor $\lambda = O(\log N)$. The second step is to find paths for the messages. Since these messages link the same processors at every step of B, it is sufficient to find paths once off-line. They can be reused over and over during the simulation. Given a set of n messages with load factor λ, it is possible to find a set of paths with congestion $c = O(\lambda + \log M)$ and dilation $d = O(\log M)$ in a fat-tree with root capacity M off-line deterministically in time polynomial in n and M. The final step is to find a set of ranks for the messages. These ranks can also be reused at each step of the simulation. Network U has root capacity $M = \Theta(\sqrt{N}/\log N)$. Thus, both the paths and the ranks for the packets can be determined off-line deterministically in time polynomial in N so that the time to simulate each step of B is $O(\log N)$.

By making minor modifications to the definition of a delay sequence, it is possible to prove that not only does the late arrival of some packet imply that a bad event occurs, but also if a bad event occurs then some packet is delayed. More precisely, some packet arrives at step $d + w$ where $w = m + qf$ if and only if there is a delay sequence of length $l \leq d + 2f - 1$ with $m + qf$ packets.

Chapter 2

Distributed random-access machines

2.1 Introduction

Underlying any realization of a parallel random-access machine (PRAM) is a communication network that conveys information between processors and memory banks. Yet in most PRAM models, communication issues are largely ignored. The basic assumption in these models is that in unit time each processor can simultaneously access one memory location. For truly large parallel computers, however, computer engineers may be hard pressed to implement networks with the communication bandwidth demanded by this assumption, due in part to packaging constraints. The difficulty of building such networks threatens the validity of the PRAM as a predictor of algorithmic performance. This chapter introduces a more restricted PRAM model, which we call a *distributed random-access machine* (DRAM), to reflect an assumption of limited communication bandwidth in the underlying network.

In a communication network, we can measure the cost of communication in terms of the number of messages that must cross a cut of the network, as in [29] and [56]. Specifically, a *cut* S of a network[1] is a subset of the nodes of the network. The *capacity* $\mathrm{cap}(S)$ is the

This chapter describes joint research with Charles Leiserson [58].

[1]We assume that in the communication network, each processor has its own local memory, the processors are interconnected as a graph, and routing of messages is performed by the processors. The generalization to the case when processors, memories, and switches are distinct entities is straightforward, but complicates the

number of wires connecting processors in S with processors in the rest of the network $\overline{S}$, i.e., the bandwidth of communication between S and $\overline{S}$. For a set M of messages, we define the *load* of M on a cut S to be the number of messages in M whose source is in S and whose destination is in $\overline{S}$ or vice versa. The *load factor* of M on S is

$$\lambda(M, S) = \frac{\text{load}(M, S)}{\text{cap}(S)},$$

and the *load factor* of M on the entire network is

$$\lambda(M) = \max_S \lambda(M, S).$$

The load factor provides a simple lower bound on the time required to deliver a set of messages. For instance, if there are 10 messages to be sent across a cut of capacity 3, the time required to deliver all 10 messages is at least the load factor 10/3.

There are two commonly occurring types of message congestion that the load factor measures effectively. One is the "hot spot" phenomenon identified by Pfister and Norton [75]. When many processors send messages to a single other processor, large delays can be experienced as messages queue for access to that other processor. In this situation, the load factor on the cut that isolates the single processor is high. The second phenomenon is message congestion due to pinboundedness. In this case, it is the limited bandwidth imposed by the packaging technology that can cause high load factors. For example, the cut of the network that limits communication performance for some set of messages might correspond to the pins on a printed-circuit board or to the cables between two cabinets.

The load-factor lower bound can be met to within a polylogarithmic factor as an upper bound on many networks, including volume and area-universal networks, such as fat-trees [29, 56], as well as the standard universal routing networks, such as the Boolean hypercube [96]. The lower bound is weak on the standard universal routing networks because every cut of these networks is large relative to the number of processors in the smaller side of the cut, but these networks may be more difficult to construct on a large scale because of packaging limitations. Networks for which the load factor lower bound cannot be approached to within a polylogarithmic factor as an upper bound include linear arrays, meshes, and high-diameter networks in general.

---

definitions.

In the PRAM model, the issue of communication bandwidth does not arise even though most parallel computers implement remote memory accesses by routing messages through an underlying network. In the PRAM model, a set of memory accesses is presumed to take unit time, reflecting the assumption that all sets of messages can be routed through the network with comparable ease. In the DRAM model, a set of memory accesses takes time equal to the load factor of the set of messages, which reflects the unequal times required to route sets of messages with different load factors.

This chapter gives DRAM algorithms that solve many graph problems with efficient communication. Our algorithms can be executed on any of the popular PRAM models because a PRAM can be viewed as a DRAM in which communication costs are ignored.

The remainder of this chapter is organized as follows. Section 2.2 contains a specification of the DRAM model and the implementation of data structures in the model. The section demonstrates how a DRAM models the congestion produced by techniques such as "recursive doubling" that are frequently used in PRAM algorithms. Section 2.3 defines the notion of a *conservative algorithm* as a concrete realization of a communication-efficient DRAM algorithm, and gives a "Shortcut Lemma" that forms the basis of the conservative algorithms in this chapter. Section 2.4 presents a conservative "recursive pairing" technique that can be used to perform many of the same functions as on lists as recursive doubling. Section 2.5 presents a linear-space exclusive-read exclusive-write conservative "tree contraction" algorithm based on the ideas of Miller and Reif [68]. Section 2.6 presents *treefix computations*, which are generalizations of the parallel prefix computation [16, 24, 71] to trees. We show that treefix computations can be performed using the tree contraction algorithm of Section 2.5. Section 2.7 gives short, efficient, parallel algorithms for tree and graph problems, most of which are based on treefix computations. Section 2.8 explores the use of concurrent reads and writes in DRAM algorithms. Section 2.9 discusses the relationship between the DRAM model and more traditional PRAM models, as well as the ramifications of using the DRAM model in practical situations.

2.2 The DRAM model

This section introduces the abstraction of a distributed random-access machine (DRAM). We show how a parallel data structure can be embedded in a DRAM, and we define the load

factor of a data structure. We show how the embedding of a data structure in a network can cause congestion in the underlying network when the pointers of the data structure are accessed in parallel, and we also demonstrate that a parallel algorithm can produce substantial congestion in an underlying network, even when there is little congestion implicit in the input data structure. We illustrate how a DRAM accurately models these two phenomena.

A DRAM consists of a set of n processors. All memory in the DRAM is local to the processors, with each processor holding a small number of $O(\lg n)$-bit registers. A processor can read, write, and perform arithmetic and logical functions on values stored in its local memory. It can also read and write memory in other processors. (A processor can transfer information between two remote memory locations through the use of local temporaries.) Each set of memory accesses is performed in a memory access *step*, and any of the standard PRAM assumptions about simultaneous reads or writes can be made. Our algorithms use only mutually exclusive memory references, however, so these special cases never arise.

The essential difference between a DRAM and a PRAM is that the DRAM models communication costs. We presume that remote memory accesses are implemented by routing messages through an underlying network. We model the communication limitations imposed by the network by assigning a numerical capacity $\mathrm{cap}(S)$ to each cut (subset of processors) S of the DRAM equal to the number of wires connecting processors in S with processors in the rest of the network. Thus, there are many different DRAM's corresponding to the many possible assignments of capacities to cuts. For a set M of memory accesses, we define $\mathrm{load}(M, S)$ to be the number of accesses in M from a processor in S to a processor in $\overline{S}$ (the rest of the DRAM), or vice versa. The load factor of M on S is $\lambda(M, S) = \mathrm{load}(M, S)/\mathrm{cap}(S)$, and the load factor of M on the DRAM is $\lambda(M) = \max_S \lambda(M, S)$.

The basic assumption in the DRAM model is that *the time required to perform a set M of memory accesses is the load factor $\lambda(M)$.* (Local operations take unit time.) This assumption constitutes the principal difference between the DRAM and the network it models. We know that the load factor is a lower bound on the time required in both the network and the DRAM. If the network's message routing algorithm cannot approach this lower bound as an upper bound (for example, if the network has high diameter), then the network is not well modeled by the DRAM. If the network's routing algorithm can nearly achieve the load factor as an upper

bound, then the analysis of an algorithm in the DRAM model will reliably predict the actual performance of the algorithm on the network. Section 2.9 discusses some networks for which the DRAM is a reasonable model, including volume-universal networks such as fat-trees [56].

A natural way to embed a data structure in a DRAM is to put one record of the data structure into each processor, as in the "data parallel" model [33]. The record can contain data, including pointers to records in other processors. We measure the quality of an embedding by treating the data structure as a set of pointers and generalizing the concept of load factor to sets of pointers. The load of a set P of pointers across a cut S, denoted $\text{load}(P,S)$, is the number of pointers in P from a processor in S to a processor in $\overline{S}$, or vice versa. The load factor of P on the entire DRAM is

$$\lambda(P) = \max_S \frac{\text{load}(P,S)}{\text{cap}(S)}.$$

The load factor of a data structure is the load factor of the set of its pointers. For many problems, good embeddings of data structures can be found in particular networks for which the DRAM is a good abstraction (see Section 2.9).

There are generally two situations in which message congestion can arise during the execution of an algorithm on a network, both of which are modeled accurately by a DRAM whose cut capacities correspond to the cut capacities of the network. In the first situation, the embedding of a data structure causes congestion because many of its pointers cross a relatively small cut of the network. A parallel access of the information across those pointers generates substantial message traffic across the cut. In the second situation, the data structure is embedded with few pointers crossing the cut, but the algorithm itself generates substantial message traffic across the cut. We now illustrate these two situations.

As an example of the first situation, consider an embedding of a simple linear list in which alternate list elements are placed on opposite sides of a narrow cut of a network. If each element fetches a value from the next element in the list, the load factor across the cut is large. In the DRAM model, this congestion is modeled by the increase in time required for the memory accesses across the cut. (Observe that in a PRAM model, the congestion is not modeled since any set of memory accesses is assumed to take unit time.) Of course, a list can typically be embedded in a network so that the number of list pointers crossing any cut is small compared to the capacity of the cut, again a situation that can be modeled by a DRAM.

In the second situation, the congestion is produced by an algorithm. As an example, consider the "recursive doubling" or "pointer jumping" technique [101] used extensively by PRAM algorithms in the literature. The idea is that each element i of a list initially has a pointer $p(i)$ to the next element in the list. At each step, element i computes $p(i) \leftarrow p(p(i))$, doubling the distance between i and the element it points to, until it points to the end of the list. This technique can be used, among other things, to compute the distance $d(i)$ of each element i to the end of the list. Initially, each element i sets $d(i) \leftarrow 1$. At each pointer-jumping step, each element i not pointing to the end of the list computes $d(i) \leftarrow d(i) + d(p(i))$. In a PRAM model, the running time on a list of length n is $O(\lg n)$. Variants of this technique are used for path compression, vertex numbering, and parallel prefix computations [68, 88, 92, 101].

We now show that recursive doubling can be expensive even when a data structure has a good embedding in a network. Figure 2-1 shows a cut of capacity 3 separating the two halves of a linked list of 16 elements. In the first step of recursive doubling, the load on the cut is only 1 because the only access across the cut occurs when element 8 accesses the data in element 9. In the second step, the load is 2 because element 7 accesses element 9 and element 8 accesses element 10. In the third step, the load is 4, and in the fourth step, each of the first eight elements makes an access across the cut, creating a load of 8. Since the load factor of the cut in the fourth step is 8/3, this set of accesses requires at least 3 time units. Whereas the capacity of the cut is large enough to support the memory accesses across it in the first step, by the fourth step, the cut capacity is insufficient. In a DRAM, this situation is modeled by the increased time to perform the memory accesses in the fourth step compared with those in the first step.

The focus of this chapter is avoiding this second cause of congestion. In Section 2.4, we shall show how a *recursive pairing* strategy can perform many of the same functions as recursive doubling, but in a communication-efficient fashion.

2.3 Conservative algorithms

This section introduces the notion of a *conservative algorithm*. In the DRAM model, a conservative algorithm is communication efficient in the sense that it never produces more congestion across cuts of the DRAM than is implicit in the input data structure. We give an important lemma that shows how pointers in a data structure can be "shortcut" without introducing

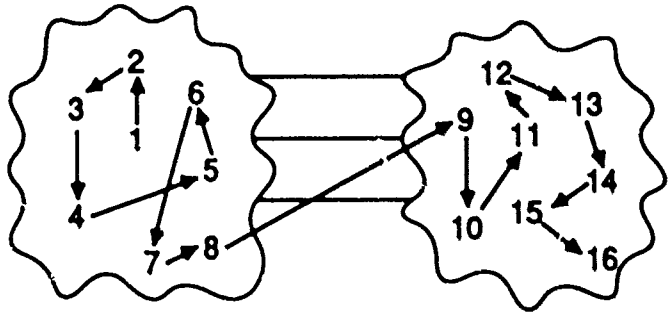

Figure 2-1: A cut of capacity 3 separating two halves of a linked list. The load of the list on the cut is 1. At the final step of recursive doubling, each element on the left side of the cut accesses an element on the right, which induces a load of 8 on the cut.

congestion.

A *conservative algorithm* is a DRAM algorithm in which the load factor of memory accesses in any step is bounded by the load factor of the input data structure, independent of the cut capacities of the DRAM on which the algorithm is executed. To be precise, we define a set M of memory accesses to be *conservative* with respect to another set M' of memory accesses if for all cuts S of a DRAM, we have $\text{load}(M, S) \leq \text{load}(M', S)$. By implication, whatever the cut capacities of the DRAM, we have $\lambda(M) \leq \lambda(M')$. We make the natural extension of the term conservative to sets of pointers and data structures. A conservative algorithm is thus one all of whose memory accesses are conservative with respect to the input data structure. Thus, if a conservative algorithm runs for T steps on an input data structure with load factor λ, then the total time for the algorithm is at most λT.

If at every step, the memory accesses of an algorithm correspond to a subset of pointers in the input data structure, then the algorithm is certainly conservative since if M is a subset of M', then we have $\text{load}(M) \leq \text{load}(M')$. For example, synchronous distributed algorithms, such as the network flow algorithms of Goldberg and Tarjan [26, 27], are conservative for this reason. We do not wish to restrict our attention to this limited class of conservative algorithms because synchronous distributed algorithms cannot efficiently solve certain problems on graphs with high diameter. For example, the problem considered earlier of determining the distance of

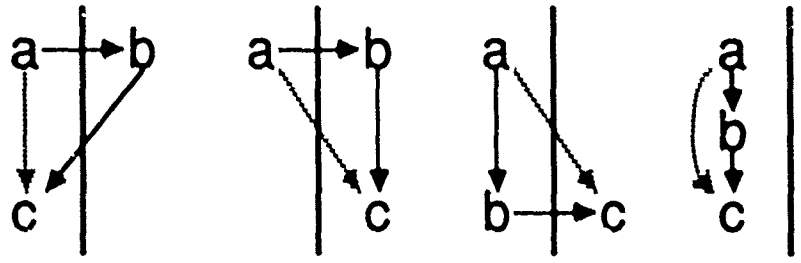

Figure 2-2: The Shortcut Lemma. In each of the four cases illustrated, the load factor across the cut is either unchanged or diminished by replacing $a \rightarrow b$ and $b \rightarrow c$ with $a \rightarrow c$.

each element to the end of the list cannot be solved in less than linear time with a synchronous distributed algorithm. A PRAM algorithm, however, can perform such the computation in logarithmic time, for example, by recursive doubling, but recursive doubling is not conservative.

We would like to know conditions under which processors in a DRAM can communicate directly with distant locations in a data structure without increasing communication requirements as measured by the load factor. The following simple, but important, lemma provides conditions that are sufficient for any DRAM.

Lemma 36 (Shortcut Lemma) *Suppose a set P of pointers in a data structure contains pointers $a \rightarrow b$ and $b \rightarrow c$. Then the set Q of pointers defined by*

$$Q = P \cup \{a \rightarrow c\} - \{a \rightarrow b, b \rightarrow c\}$$

is conservative with respect to P. Moreover, any set Q of pointers is conservative with respect to another set P of pointers if there exist pointer-disjoint paths in P that connect the endpoints of pointers in Q.

Proof: We show only the first part of the lemma since the second part follows immediately by induction. We shall show that $\text{load}(Q, S) \leq \text{load}(P, S)$ for any cut S of the DRAM. Consider the eight ways in which a, b, and c can be assigned to sides of the partition induced by a cut S. Half the cases can be eliminated by symmetry if we assume that a is on the left side. In each of the four remaining cases, the load across the cut is either unchanged or diminished when $a \rightarrow b$ and $b \rightarrow c$ are replaced with $a \rightarrow c$, as is shown in Figure 2-2. □

In summary, this section has introduced the notion of a conservative algorithm. An upper bound on the time required by a conservative algorithm can be determined solely from the embedding of an input data structure on the DRAM. If the number of steps of the conservative algorithm is T and the load factor of the input data structure is λ, then the total time is at most λT. A user of a conservative algorithm therefore need only minimize the congestion of pointers in the input data structure across cuts of the DRAM to minimize the time required by the algorithm. If the embedding of the data structure is good, that is, its load factor is small, then a conservative algorithm that uses a small number of steps runs fast.

2.4 List contraction

In this section we present a conservative "recursive pairing" algorithm, Algorithm LC, that can perform many of the same functions on lists as recursive doubling. The idea is to *contract* an input list by repeatedly pairing and merging adjacent elements of the list until only a single element remains. The merges are recorded as internal nodes of a binary *contraction tree* whose leaves are the elements in the input list. After building the contraction tree, operations such as broadcasting from the root or parallel prefix can be performed in a conservative fashion. Algorithm LC is a randomized algorithm, and with high probability, the height of the contraction tree and the number of steps on a DRAM are both $O(\lg n)$, where n is the number of elements in the input list. A deterministic variant based on deterministic coin tossing [20] runs in $O(\lg n \lg^* m)$ steps, where m is the number of processors in the DRAM, and produces a contraction tree of height $O(\lg n)$.

The recursive pairing strategy is illustrated in Figure 2-3 for a list (A, B, C, D, E). In the first step, elements B and C pair and merge, as do elements D and E. The merges are shown as contours in the figure. A new *contracted* list (A, BC, DE) is formed from the unpaired element A and the two compound elements BC and DE. After the second step of the algorithm, the contracted list consists of the elements ABC and DE. The third and final step reduces the list to the single element $ABCDE$.

In Algorithm LC, the contours of Figure 2-3 are represented in a data structure called a *contraction tree*. The leaves of the contraction tree are the list elements, and the internal nodes are the contours. To maintain the contraction-tree data structure, the algorithm requires

constant extra space for each element in the input list. Each processor contains two elements: an element in the input list, and a *spare* element that will act as an internal node in the contraction tree. We call the two elements in the same processor *mates*. Each element holds a pointer to an unused internal node, which for each list element initially points to its mate. The use of spare nodes allows the algorithm to distribute the space for the internal nodes of the contraction tree uniformly over the elements in the list. (Spare internal nodes are used in [14] and [55] for similar reasons, but in a different context.)

We now describe the operation of Algorithm LC, which is illustrated in Figure 2-4 for the example of Figure 2-3. (A description in pseudocode can be found in [57].) In the first step, each element of the input list randomly picks either its left or right neighbor. An element at the left or right end of the list always picks its only neighbor. If two elements pick each other, then they merge. The merge is recorded by making the spare of the left element of the pair be the root of a new contraction tree. The spare of the right element becomes the spare for the root, and the elements themselves become the children of the root. The roots of the new contraction trees and the unpaired list elements now form themselves into a new list representing the contracted list, upon which the algorithm operates recursively.

At each step of the algorithm, any given element of the contracted list is a set of consecutive elements in the input list—a contour in Figure 2-3. The set is represented by a contraction-tree data structure whose leaves are the elements of the set and whose internal nodes record the merges. When the entire input list has been contracted to a single node, the algorithm terminates and a single contraction tree records all of the merges.

To describe the efficiency of randomized algorithms such as Algorithm LC, we shall sometimes say that an algorithm runs in $O(T(n))$ steps "with high probability," by which we shall mean that for any constant $k > 0$, there are constants $c_1 > 0$ and $c_2 > 0$ such that with probability $1 - c_1/n^k$, the algorithm terminates in at most $c_2 T(n)$ steps.

Theorem 37 *With high probability, Algorithm LC takes $O(\lg n)$ steps to construct a contraction tree for a list of n elements.*

Proof: We show that the algorithm terminates after $(k+1)\log_{4/3} n$ iterations with probability at least $1 - 1/n^k$. We use an accounting scheme involving "tokens" to analyze the algorithm.

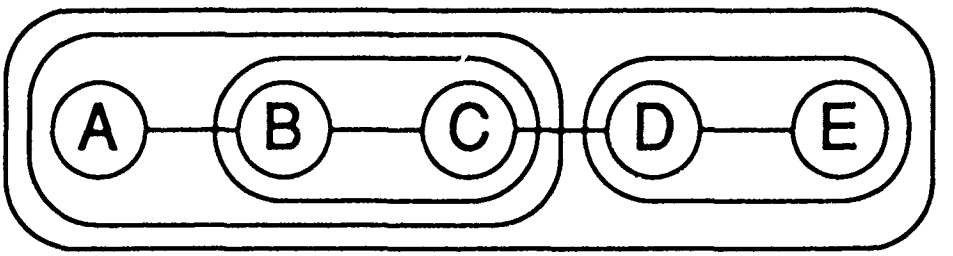

Figure 2-3: The recursive pairing strategy operating on a list (A, B, C, D, E). Merged nodes are shown as contours, and the nesting of contours gives the structure of the contraction tree.

Initially, a unique token resides between each pair of elements in the input list. Whenever two list elements pick each other, we destroy the token between them. For each token destroyed, the length of the list decreases by one, and the algorithm terminates when no token remains. In any iteration, an existing token has probability at least 1/4 of being destroyed. Thus, after m iterations, a token has probability at most $(3/4)^m$ of remaining in existence. Let T_i be the event that token i exists after m iterations, and let T be the event that any token remains after m iterations. Then the probability that any token remains after m iterations is given by

$$\begin{aligned}
\Pr\{T\} &= \Pr\{T_1 \cup T_2 \cup \ldots \cup T_{n-1}\} \\
&\leq \Pr\{T_1\} + \Pr\{T_2\} + \cdots + \Pr\{T_{n-1}\} \\
&\leq (n-1)\left(\frac{3}{4}\right)^m .
\end{aligned}$$

For $m = (k+1)\log_{4/3} n$ iterations, we have

$$\begin{aligned}
\Pr\{T\} &\leq (n-1)\left(\frac{3}{4}\right)^{(k+1)\log_{4/3} n} \\
&\leq \frac{1}{n^k} .
\end{aligned}$$

□

Theorem 38 *With high probability, a contraction tree constructed by Algorithm LC on a list of n elements has height $O(\lg n)$.*

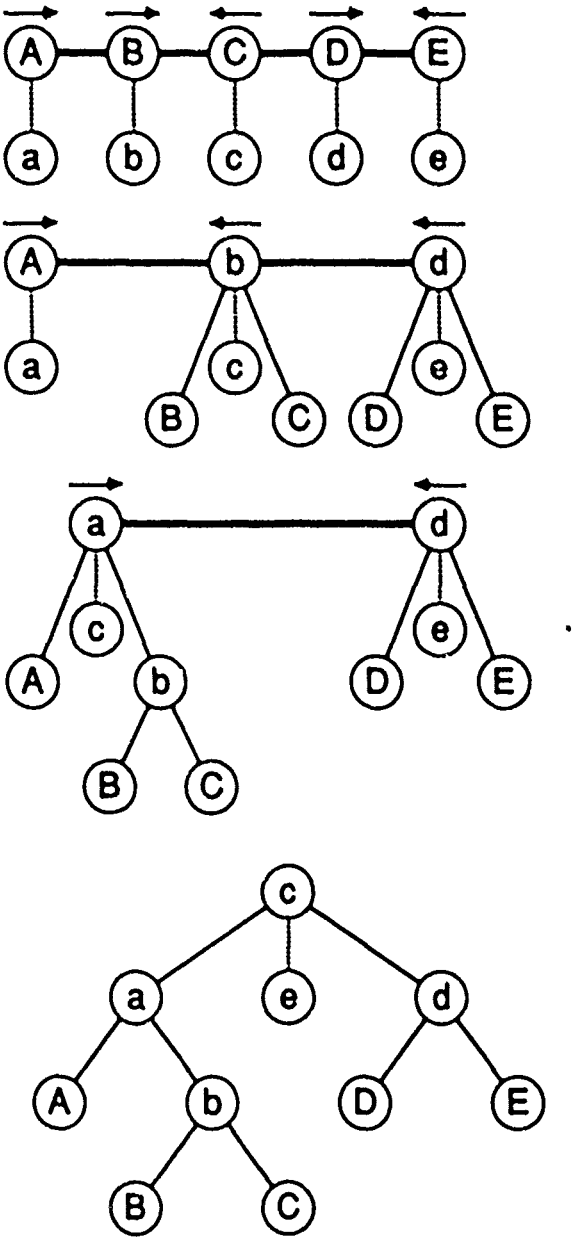

Figure 2-4: The operation of Algorithm LC on the example of Figure 2-3. The input list is (A, B, C, D, E), and the corresponding spares are in lower case. When elements B and C pair and merge in the first step of the algorithm, the spare b becomes the root of a contraction tree with leaves B and C to represent the compound node BC. The spare for b is c. At the end of the first step, the list consisting of the elements A, b, and d represents the contracted list (A, BC, DE). After two more contraction steps of Algorithm LC, the input list is contracted to a single element $ABCDE$, which is represented by a contraction tree whose root is c and whose leaves are the elements of the input list (A, B, C, D, E).

Proof: The height of the contraction tree is no greater than the number of iterations of Algorithm LC. □

We now prove that Algorithm LC is conservative.

Theorem 39 *Algorithm LC is conservative.*

Proof: By convention, let the mate of an element in the input list lie in the order between that element and its right neighbor. The key idea is that the order of the list elements and their spares is preserved by the merging operation, and consequently, after each contraction step, the pointers in the contracted list correspond to disjoint paths in the original list, and the pointers between elements and their spares also correspond to disjoint paths. By the Shortcut Lemma these two sets of pointers are each conservative with respect to the input list, and since each set of memory accesses in a contraction step of the algorithm is a subset of one of these two sets, the algorithm is conservative. □

Although a contraction tree itself is not conservative with respect to an input list, it can be used as a data structure in conservative algorithms. For example, contraction trees can be used to efficiently broadcast a value to all of the elements of a list and to accumulate values stored in each element of a list.

More generally, contraction trees are useful for performing *prefix computations* in a conservative fashion. Let $\mathcal{D}$ be a domain with a binary associative operation $\cdot$ and an identity ε. A prefix computation [16, 24, 71] on a list with elements $x_1, x_2, \ldots, x_n$ in $\mathcal{D}$ puts the value $y_i = x_1 \cdot x_2 \cdots x_i$ in element i for each $i = 1, 2, \ldots, n$.

A prefix computation on a list can be performed by a conservative, two-phase algorithm on the contraction tree. The leaves of the contraction tree from left to right are the elements in the list from x_1 to x_n. The first phase proceeds bottom up on the tree. Each leaf passes its x value to its parent. When an internal node receives a value z_l from its left child and a value z_r from its right child, the node saves the value z_l and passes $z_l \cdot z_r$ to its parent. When the root receives values from its children, the second top-down phase begins. The root passes ε to its left child and its z_l value to its right child. When an internal node receives a value z_p from its parent, it passes z_p to its left child, and passes $z_l \cdot z_p$ to its right child. When a leaf receives z_p it computes $y = z_p \cdot x$.

The number of steps required by the prefix computation is proportional to the height of the tree, which with high probability is $O(\lg n)$. At each step, the algorithm communicates across a set of pointers in the contraction tree, all of which are the same distance from the leaves in the first phase, and the same distance from the root in the second. That this computation is performed in a conservative fashion is a consequence of the following lemma.

Theorem 40 *Let CT be a contraction tree computed by Algorithm LC on an input list L, and suppose P is a subset of the pointers of CT. If no pointer in P is an ancestor of another pointer in P, then P is conservative with respect to L.*

Proof: An inorder traversal of CT alternately visits list elements (leaves) and their mates (internal nodes) in the same order that the list elements and mates appear in L. Thus, if no pointer in P is an ancestor of another pointer in P, the pointers in P correspond to disjoint paths in L. By the Shortcut Lemma, any set of pointers that correspond to disjoint paths in the list L are conservative with respect to L. □

Algorithm LC, which constructs a contraction tree in $O(\lg n)$ steps, is a randomized algorithm. By using the "deterministic coin tossing" technique of Cole and Vishkin [20], the algorithm can be performed nearly as well deterministically Specifically, the randomized pairing step can be performed deterministically in $O(\lg^* m)$ steps on a DRAM with m processors, where $\lg^* m$ is the number of times the logarithm function must be successively applied to reduce m to a value at most 1. The overall running time for list contraction is thus $O(\lg n \lg^* m)$.

As a final comment, we observe that with minor modifications, Algorithm LC can be used to contract circular lists with the same complexity bounds as for linear lists.

2.5 Tree contraction

This section presents a conservative tree contraction algorithm, Algorithm TC, based on the tree contraction ideas of Miller and Reif [68]. The algorithm uses a recursive pairing strategy to build a contraction tree for an input binary tree in much the same manner as Algorithm LC does for a list. With high probability, the height of the contraction tree and the number of steps on a DRAM are both $O(\lg n)$, where n is the number of nodes in the input tree. A deterministic variant runs in $O(\lg n \lg^* m)$ steps and produces a contraction tree of height $O(\lg n)$.

The recursive pairing strategy for trees is illustrated in Figure 2-5 for a tree with nodes A, B, C, D, E, and F. In the first step nodes A and B pair and merge, as do nodes C and D; the merges are shown as contours in the figure. A new *contracted* tree is formed from the unpaired nodes E and F, and the compound nodes AB and CD. In the next step of the algorithm, node E pairs and merges with CD to form a node CDE. After two more steps the 6-node input tree has been contracted to a single node. Notice that each node shown as a contour in the figure is a connected subgraph of the input tree, and that the node has at most two children in the contracted tree.

Algorithm TC represents the contours of Figure 2-5 in a contraction-tree data structure in the same manner as Algorithm LC represents the contours of Figure 2-3. Space for the internal nodes of the contraction tree is again provided by spares. Initially, the spare of each node in the input tree is its mate, an unused node stored in the same processor.

We now outline Algorithm TC in more detail. (A description in pseudocode can be found in [57].) In the first step, nodes in the input tree are paired. The pairing strategy has each node pick from among its neighbors according to how many children it has. A leaf picks its parent with probability 1. A node with exactly one child picks either its child or its parent, each with probability 1/2. A node with two children picks either child, each with probability 1/2. The root, which has no parent, picks its children with equal probability. If two nodes pick each other, then they merge. The merge is recorded by making the spare of the parent in the pair be the root of a new contraction tree. The spare of the child in the pair becomes the spare for the root, and the parent and child themselves become the children of the root. The new nodes and the unpaired nodes form themselves into a new tree that represents the contracted tree, upon which the algorithm operates recursively. The contracted tree is binary because the pairing strategy ensures that no node with two children pairs with its parent.

In the next section, we shall need to *expand* a contracted tree in order to describe treefix computations recursively. Expansion consists of undoing the merges in the reverse of the order in which they occurred. From the time that a parent and child merge to the time that the node representing their merge in the contraction tree expands, the pointers of the pair are undisturbed. Consequently, these pointers can be used to restore the pointers of the neighbors of the pair to the state they had immediately before the pair merged. To ensure that the merges

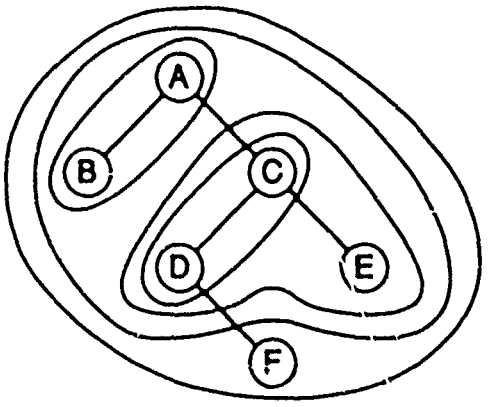

Figure 2-5: The recursive pairing strategy operating on a tree with nodes A,B,C,D,E, and F. Merged nodes are shown as contours, and the nesting of contours gives the structure of the contraction tree.

are undone in the exact reverse order, as is assumed in the next section, it is helpful to store in each internal node of the contraction tree the step number in which the merge took place. In fact, the tree can be expanded by a greedy strategy without consulting the number of the contraction step at which each merge occurred.

The proof that with high probability, Algorithm TC takes $O(\lg n)$ steps to contract an input binary tree to a single node requires three technical lemmas. The first lemma shows that in a binary tree, the number of nodes with two children and the number of leaves are nearly equal. The second lemma provides an elementary bound on the expectation of a discrete random variable with a finite upper bound. The last lemma presents a Chernoff-type bound [18] on the tail of a binomial distribution.

Lemma 41 *Suppose $T = (V, E)$ is a rooted binary tree, and let V_0, V_1 and V_2 denote the sets of nodes in T (excluding the root), with zero, one, or two children, respectively, and let $d(r)$ be the degree of the root. Then we have*

$$|V_0| = |V_2| + d(r).$$

□

Lemma 42 *Let $X \leq b$ be a discrete random variable with expected value μ. For $w < b$, we have*

$$\Pr\{X \geq w\} \geq \frac{\mu - w}{b - w}.$$

□

The final lemma presents a bound on the tail of a binomial distribution. Consider a set of t independent Bernoulli trials, each occurring with probability p of success. The probability that fewer than s successful trials occur is

$$B(s,t,p) = \sum_{k=0}^{s-1} \binom{t}{k} p^k (1-p)^{t-k}.$$

The lemma bounds the probability $B(s,t,p)$ that fewer than s successes occur in t trials when $s < t/2$ and $p < 1/2$.

Lemma 43 *For $s < t/2$ and $p < 1/2$, we have*

$$B(s,t,p) \leq \left(\frac{1-p}{1-2p}\right)(1-p)^t \left(\frac{et}{s}\right)^s.$$

□

With these lemmas we can now prove that with high probability, Algorithm TC takes $O(\lg n)$ steps to contract a rooted binary tree to a single node.

Theorem 44 *With high probability, Algorithm TC takes $O(\lg n)$ contraction steps to contract a rooted binary tree of n nodes to a single node.*

Proof: The proof has three parts. First, we use Lemma 41 to show that if a rooted binary tree has $|V|$ nodes, the expected number of nodes pairing with a parent in a single contraction step is at least $|V|/4$. Next, we use Lemma 42 to show that the probability that at least $|V|/8$ nodes pair with a parent in any step is at least 1/3. Finally, we use Lemma 43 to show for any constant k, that after $\alpha \log_{8/7} n$ steps, for some constant $\alpha > 2$, the probability that the tree has not contracted into a single node is $O(1/n^k)$.

We first show that the expected number of nodes pairing with a parent is at least $|V|/4$, provided that $|V| > 1$. A child is picked by its parent with probability 1 when its parent is a degree-1 root, and 1/2 otherwise. Thus, a leaf pairs with its parent with probability at least 1/2, and a node (other than the root) with one child pairs with its parent with probability at least 1/4. Let P be the number of nodes pairing with a parent. Then we have

$$\mathrm{E}(P) \geq \frac{|V_0|}{2} + \frac{|V_1|}{4},$$

and applying Lemma 41 yields the desired result:

$$\mathrm{E}(P) \geq \frac{|V_0| + |V_1| + |V_2| + d(r)}{4} \geq \frac{|V|}{4}.$$

Now we show that the probability that at least $|V|/8$ nodes pair with a parent in a single contraction step is at least 1/3. We call such a step *successful*. At most half of the nodes pair with their parents. Using Lemma 42 with $b = |V|/2$, $w = |V|/8$, and $\mu \geq |V|/4$, we have

$$\Pr\left\{P \geq \frac{|V|}{8}\right\} \geq \frac{\frac{|V|}{4} - \frac{|V|}{8}}{\frac{|V|}{2} - \frac{|V|}{8}} = \frac{1}{3}.$$

Finally, we show that with high probability, Algorithm TC takes $O(\lg n)$ contraction steps to contract the input tree to a single node. The size of the tree after a contraction following a successful pairing step is at most 7/8 the size before the contraction. After $\log_{8/7} n$ successful steps, the tree must consist of a single node. By Lemma 43, the probability that fewer than $\log_{8/7} n$ successful steps occur in $\alpha \log_{8/7} n$ steps is

$$\begin{aligned} B(\log_{8/7} n, \alpha \log_{8/7} n, 1/3) &\leq 2((2/3)^{\alpha} \alpha e)^{\log_{8/7} n} \\ &= 2n^{\log_{8/7}((2/3)^{\alpha} \alpha e)}. \end{aligned}$$

For any value k, we can choose α large enough so that $B(\log_{8/7} n, \alpha \log_{8/7} n, 1/3) = O(1/n^k)$. In particular, for $k = 1$ a value of $\alpha = 8$ suffices. □

We now prove that Algorithm TC is communication efficient on the DRAM model.

Theorem 45 *Algorithm TC is conservative.*

Proof: Each node of a contracted tree is a connected subgraph of the input tree. The root of the contraction tree that represents the subgraph is called the *representative* of the subgraph.

The representative and its spare are each either a node of the subgraph or a mate of a node of the subgraph.

Every set of memory accesses performed by the algorithm is of one of two types. In the first type, each representative of a subgraph communicates with its spare, if at all. In the second type, each representative of a subgraph communicates with the representative of one of its children in the contracted tree. In either of these two cases, the set of memory accesses corresponds to a set of disjoint paths in the input graph, and hence, by the Shortcut Lemma, is conservative with respect to the input graph. □

Tree contraction can be performed conservatively and deterministically on a DRAM with m processors in $O(\lg n \lg^* m)$ steps using the deterministic coin-tossing algorithm of Cole and Vishkin [20]. The key idea is that in Algorithm TC, the nodes that can pair form chains, and by Lemma 41 these chains contain at least half the tree edges. The chains can be oriented from child to parent in the tree, and deterministic coin tossing can be used to perform the pairing step in $O(\lg^* m)$ steps.

2.6 Treefix computations

This section presents a generalization of the parallel prefix computation to binary trees. We present two kinds of *treefix* computations—*rootfix* and *leaffix*—and show how they can be implemented by an $O(\lg n)$-step conservative algorithm that uses $O(n)$ space, where n is the number of nodes in the input tree. As we shall see in Section 2.7, treefix computations can greatly simplify the description of many parallel graph algorithms in the literature, and moreover, treefix computations can be performed by conservative algorithms.

We begin with a definition of treefix computation.

Definition. Let $\mathcal{D}$ be a domain with a binary associative operation $\cdot$ and an identity ε. Let T be a rooted, binary tree in which each vertex $i \in T$ has an assigned input value $x_i \in \mathcal{D}$. The *rootfix* problem is to compute for each vertex $i \in T$ with parent j, the output value $y_i = y_j \cdot x_i$, where $y_j = \varepsilon$ if i is the root. The *leaffix* problem is to compute for each vertex $i \in T$ with left child j and right child k, the output value $y_i = x_i \cdot y_j \cdot y_k$, where $y_j = \varepsilon$ if i has no left child

and $y_k = \varepsilon$ if i has no right child.

Simple examples of treefix problems are computing the depth of each vertex in a rooted binary tree and computing the size of each subtree. These and other examples appear in the next section.

Like the prefix computation on lists, treefix computations can be performed directly on the contraction tree. For simplicity, however, we describe a recursive version.

Theorem 46 *Let T be a binary tree of n nodes on a DRAM with m processors. A rootfix or leaffix computation can be performed on T by a conservative randomized algorithm which, with high probability, takes $O(\lg n)$ steps, or by a conservative deterministic algorithm which takes $O(\lg n \lg^* m)$ steps. Both algorithms use $O(1)$ space per node of the tree.*

Proof: Both treefix computations are performed by executing a single contraction step on the input tree T to produce a contracted tree T'. Each node in T' is assigned an input value, and the treefix computation is executed recursively on T'. The contracted tree T' is then expanded to yield T once again, and the output value of each node in T is computed from the input values of T and the output values of T'.

The algorithm for leaffix is based on each node i maintaining a value s_i which has the form $a_i \sqcup b_i \sqcup c_i$, where $a_i, b_i, c_i \in \mathcal{D}$ are elements of the domain, and the character "$\sqcup$" represents symbolically a slot to be filled in with a value. The number of slots is equal to the number of children of the node, and each slot corresponds to a specific child. When a parent and child pair during the course of the leaffix algorithm, the value of the child is substituted into the corresponding slot in the value of its parent. For example, suppose node i pairs with its right child j, where the value of i is $s_i = a_i \sqcup b_i \sqcup c_i$ and the value of j is $s_j = a_j \sqcup b_j$. The value s_k of the merged node k is computed from s_i and s_j by substituting s_j into the second slot in s_i, yielding the value $s_k = a_i \sqcup b_i \cdot a_j \sqcup b_j \cdot c_i$. The $\cdot$ operations are carried out immediately so that s_k has the proper form.

The leaffix algorithm initializes each node i by $s_i \leftarrow x_i$, $s_i \leftarrow x_i \sqcup$, or $s_i \leftarrow x_i \sqcup \sqcup$ depending on the number of children of node i. The algorithm then proceeds as follows. At the end of a contraction step, each node k in T' that results from the merging of parent i and child j

computes its value s_k by substituting s_j into the appropriate slot of s_i. The leaffix algorithm is then performed recursively on T' using the s values as inputs and yielding y values as output. (The y values contain no slots and are simply elements of the domain $\mathcal{D}$.) During the expansion step, the parent node i sets $y_i \leftarrow y_k$. Each child node j gets its output value y_j by substituting the y values of its children into the slots of s_j.

In the rootfix algorithm, each node i maintains a value s_i, as in the leaffix algorithm, but each s_i now has the general form $s_i = \sqcup a_i$, and the slot of a node corresponds to the node's parent. The rootfix algorithm initializes each node i by $s_i \leftarrow \sqcup x_i$, except for the root r which performs $s_r \leftarrow x_r$. After the pairs have been determined for the contraction step, each node j that is the child in a pair, and which itself has a child, substitutes s_j in the appropriate slot of its child's value. At the end of a contraction step, each node k in T' that resulted from the merging of parent i and child j computes its value by $s_k \leftarrow s_i$. The rootfix algorithm is then performed recursively on T', yielding y values as output. During the expansion step, the parent node i sets $y_i \leftarrow y_k$. Each child node j gets its output value y_j by substituting y_i into the slot of s_j.

The time and space bounds claimed in the theorem are apparent by inspection. Each step of a treefix algorithm adds only a constant amount of work to a corresponding step in the tree contraction and expansion algorithms. The additional space required by the treefix algorithms is the $O(1)$ space per node for the x, y, and s values. □

2.7 Graph algorithms

This section presents a collection of conservative DRAM algorithms for solving graph problems. The algorithms use two processors per edge of an input graph $G = (V, E)$ and require only constant extra space in each processor. Most of the algorithms use treefix computations as subroutines.

We represent each vertex in an undirected graph $G = (V, E)$ by a doubly linked *incidence ring* of processors, one for each edge. Each element of the incidence ring contains pointers to the next and previous elements in the ring, and one pointer for a graph edge. For each edge $(u, v) \in E$, the element in the incidence ring for u contains a pointer to an edge element in the

incidence ring for v, and vice versa. A directed graph is represented in the same doubly linked fashion, but the graph edges are labeled with their directions.

We represent trees with arbitrary vertex degrees by an incidence ring structure as well. If the tree is directed, each ring has a unique *principal element* that points toward the root. Breaking the incidence ring before the principal element yields the standard binary-tree representation of the tree [39, pp. 332–333].

We now present brief descriptions of the algorithms. The performance is given in terms of the number of steps on a DRAM when the input representation has size n. We assume the implicit tree contractions in the algorithms are performed by the randomized Algorithm TC. Deterministic bounds can be obtained by multiplying the number of steps by $O(\lg^* m)$, where m is the number of processors. An upper bound on the time required in the DRAM model can be obtained by multiplying the number of steps by the load factor of the input.

Generalized treefix. *Perform a treefix operation on a directed tree with arbitrary vertex degree. The input values $\{x_i\}$ are stored in the principal elements of the tree, which is where the output values $\{y_i\}$ are to be placed. The leaffix value at a node i whose children have values $y_1, y_2, \ldots, y_k$ is $y_i = x_i \cdot y_1 \cdot y_2 \cdots y_k$.* Each non-principal element is assigned the identity ϵ for its value. A binary treefix computation performed on the binary tree representation underlying the tree computes the desired values. *Performance:* $O(\lg n)$.

Tree functions. *Given a directed tree, compute for each node the number of descendants, its height, or its depth.* The number of descendants for each node can be computed by a leaffix computation with $\cdot$ as integer addition and $x_i = 1$ for all nodes. The height of a node can also be computed by a leaffix computation where $a \cdot b = \max(a+1, b+1)$, the identity is $\epsilon = -1$, and $x_i = -1$ for all nodes.[2] The depth of a node can be computed by a rootfix computation with $\cdot$ as addition and $x_i = 1$ for all nodes except the root which has value 0. *Performance:* $O(\lg n)$.

Rooting an undirected tree. *Pick a root of a tree with undirected graph pointers, and orient the graph pointers toward the root.* Form an "Eulerian tour" of the pointers of the representation [92] by directing each element of the tree to link its incoming ring pointer with

[2]Technically, $\epsilon = -1$ is not an identity for the operation $a \ \ b = \max(a+1, b+1)$. Nonetheless, this leaffix computation correctly computes the height of each node in a binary tree. Moreover, this leaffix computation also generalizes to a directed tree with arbitrary vertex degree.

its graph edge directed outward and its graph edge directed inward with its outgoing ring pointer. Each graph edge is used twice in the tour, once in each direction, but each ring pointer is used only once. Use Algorithm LC to form a contraction tree of the tour. Choose the root of the contraction tree to be the root of the tree, and break the tour so that it begins with the root. Use parallel prefix to number each node according to its first occurrence in the tour. Use contraction trees to distribute the smallest value in each incidence ring to the elements of the ring. Orient each graph edge from the larger value to the smaller. *Performance:* $O(\lg n)$.

Rerooting a directed tree. *Given a directed tree and another distinguished vertex k, reorient the graph edges of the tree to point toward k.* The algorithm for rooting a tree can be used by picking k as the root instead of the root of the contraction tree, but a single treefix computation suffices. Perform a leaffix computation with $x_k = 1$ and $x_i = 0$ if $i \neq k$, and use Boolean OR for $\cdot$. Each principal element whose leaffix value is 1 lies on the path from x_k to the root. Reverse the direction of the graph pointers of these elements. (Note: rerooting a tree changes the principal elements.) *Performance:* $O(\lg n)$.

Tree-walk numberings of a binary tree. *Number the nodes of a binary tree according to the order they would be visited in a preorder/inorder/postorder tree walk.* For each of the walks, we compute y_k, the number of nodes visited before the left subtree of k. Use a leaffix computation to compute the number $size_k$ of the subtree rooted at k. We first compute the preorder numbering. (For the purposes of these numbering algorithms, we consider the root to be a left child.) If node k is a left child, set $x_k \leftarrow 1$. If node k is a right child, set x_k to the size of its sibling subtree plus 1. A rootfix computation with $\cdot$ yields y_k, which is the preorder numbering of node k. The inorder numbering can be computed similarly. If node k is a left child, set $x_k \leftarrow 0$. If k is a right child, set x_k to the size of its sibling subtree plus 1. Compute y_k for each node using a rootfix computation with $+$. The inorder numbering of node k is y_k plus the size of its left subtree plus 1. The postfix numbering can be computed by setting $x_k \leftarrow 0$ if node k is a left child, and by setting x_k to the size of its sibling subtree if k is a right child. After computing y_k using a rootfix computation with $+$, the postfix numbering of node k is y_k plus the sizes of its two subtrees plus 1. *Performance:* $O(\lg n)$.

Prefix and postfix numberings of a directed tree. *Number the edges of an arbitrary directed tree according to the order they are visited in a preorder/postorder tree walk.* The

problem reduces to prefix/postfix numbering on the underlying binary tree representation. *Performance:* $O(\lg n)$.

Diameter and center of a tree. *The diameter is the length of the longest path in the tree. A center is a vertex v such that the longest path from v to a leaf is minimal over all vertices in the tree.* The diameter can be determined by rooting the tree and using rootfix to find the farthest leaf from the root. Reroot the tree at this leaf. The distance from the new root to the farthest leaf is the diameter. (This algorithm is based on an analog algorithm attributed to J. Wennmacker [23].) A center of the tree can be determined by finding a median element of the path that realizes the diameter. *Performance:* $O(\lg n)$.

Centroid of a tree. *A centroid is a vertex v such that the largest subtree with v as a leaf is minimal over all vertices in the tree.* A centroid can be determined by rooting the tree and computing the size of each subtree. By broadcasting the size m of the tree from the root, each graph edge in each incidence ring can determine the number of elements on the other side of the edge. For each incidence ring, compute the maximum of these values. A vertex with the minimum of these maximum values is a centroid. *Performance:* $O(\lg n)$.

Separator of a tree. *A separator* [62] *is a partition of the vertices of an n-vertex tree into three sets A, B, and C, with $|A| \leq \frac{2}{3}n$, $|B| = 1$, and $|C| \leq \frac{2}{3}n$, such that no edge of the tree goes between a vertex in A and a vertex in C.* Determine a centroid of the tree. This vertex is the separator vertex in B. It remains to partition the remaining vertices between A and C. For each graph edge in the incidence ring, count the number of vertices in the subtree on the other side of the edge. Put the largest subtree in A. Use parallel prefix on the incidence ring to compute a running sum of the sizes of the other subtrees. Put all subtrees whose prefix value is at most $\frac{2}{3}n$ in C, and put the remainder in A. *Performance:* $O(\lg n)$.

Subexpression evaluation. *Given a directed tree in which each leaf has a value and each internal node has an operator from $\{+, -, \cdot, \div\}$, compute for each internal node the subexpression rooted at that node.* A single leaffix-like computation suffices using the ideas of Brent [15] and Miller and Reif [68]. *Performance:* $O(\lg n)$.

Minimum-cost spanning forest. *A spanning forest of an undirected graph $G = (V, E)$ is a maximal set $F \subseteq E$ of edges that contains no cycles. Given an undirected graph $G = (V, E)$ and a cost function $w : E \to \mathbf{R}$, determine a spanning forest F such that the sum of the costs*

(weights) of the edges in F is minimum. We give a conservative DRAM implementation of Boruvka's algorithm, also attributed to Sollin [91, pp. 71–83]. We assume without loss of generality that the edge weights are distinct—otherwise, we can assign the weight of a graph edge e between two incidence-ring elements with addresses a and b to be $(w(e), \max(a,b), \min(a,b))$ and then compare weights lexicographically. We determine F by marking edges in G. Initially, no edges are marked. At each step of the algorithm, the currently marked graph edges form a subforest of F. Break each incidence ring by removing a single ring pointer, and direct the resulting linear list. At each step of the algorithm, the marked graph edges and the ring pointers form a set $\{T_i\}$ of rooted trees, where the index i of the tree is the address of the root. The algorithm proceeds as follows. For each tree T_i, use a rootfix computation to broadcast i to all of the elements in T_i. Use a leaffix computation on T_i to determine an edge $e \in E$ connecting an edge element $u \in T_i$ with an edge element $v \in T_j$, where $i \neq j$, with smallest weight. If no such edge exists, the algorithm terminates. If T_j picks the same edge as T_i, the tree with smaller index does nothing. Otherwise, mark e as a member of F, directing it into T_j, and reroot T_i with u as the new root. Repeat this procedure until the algorithm terminates. *Performance:* $O(\lg^2 n)$.

Connected components. *Given an undirected input graph $G = (V, E)$, determine a labeling $l : V \to \mathbb{Z}$ such that $l(v) = l(v')$ if and only if v and v' are in the same connected component of G.* The algorithm is the same as the minimum spanning tree algorithm, choosing the weight of a graph edge e between incidence ring elements with addresses a and b to be $(\max(a,b), \min(a,b))$. The label of a vertex is the index of its tree. *Performance:* $O(\lg^2 n)$.

Biconnected components. *Two edges of an undirected graph $G = (V, E)$ are in the same biconnected component if they lie on a common simple cycle. Determine a labeling $l : E \to \mathbb{Z}$ such that $l(e) = l(e')$ if and only if e and e' are in the same biconnected component of G.* We give a conservative DRAM implementation of the biconnectivity algorithm of Tarjan and Vishkin [92]. We assume that the reader has some familiarity with that algorithm. Find a (directed) minimum spanning tree $T = (V, F)$ of G. Number the vertices in the minimum spanning tree in preorder. Use leaffix computations to compute for each vertex v three values: $\mathrm{nd}(v)$, $\mathrm{low}(v)$, and $\mathrm{high}(v)$. The value $\mathrm{nd}(v)$ is the number of descendants of v, and $\mathrm{low}(v)$ ($\mathrm{high}(v)$) is the lowest (highest) numbered vertex (with respect to the preorder numbering of T) that is either

a descendant of v or adjacent to a descendant of v by an edge of $E - F$. Build a new graph G' where the edges of F are the vertices of G'. Let e be an edge from u to $p(u)$, where $p(u)$ is the parent of u in F. The adjacency ring for u in G acts as the adjacency ring for e in G'. Add two kinds of edges to G'. For each edge $\{w, v\}$ in $E - F$ such that $v + \mathrm{nd}(v) \leq w$, add an edge $\{\{v, p(v)\}, \{w, p(w)\}\}$ to G'. For each edge $(v, p(v))$ of F such that $v \neq 1$ and $p(v) \neq 1$, and $\mathrm{low}(v) < v$ or $\mathrm{high}(v) \geq p(v) + \mathrm{nd}(p(v))$, add an edge $\{\{v, p(v)\}, \{p(v), p(p(v))\}\}$ to G'. It can be verified that the representation of G' is conservative with respect to the representation of G. Find the connected components of G'. Two edges of F are in the same block if as vertices in G' they are in the same connected component. Finally, for each edge $e = \{w, v\}$ in $E - F$, let $l(e) = l(\{w, p(w)\})$. *Performance:* $O(\lg^2 n)$.

Eulerian cycle. *An Eulerian cycle of an undirected graph $G = (V, E)$ is a cycle containing each edge in E exactly once.* If any vertex has odd degree, then no Eulerian cycle exists. Form a set of disjoint cycles of the pointers of the representation of G as in the algorithm for directing a tree. The cycles can be merged using an algorithm similar to the minimum-cost-spanning-forest algorithm [5, 7]. *Performance:* $O(\lg^2 n)$.

2.8 Concurrent reads and writes

This section explores the use of concurrent reads and writes to memory in a DRAM. When concurrent reads and writes are allowed, the definition of load must be modified so that the load factor remains a lower bound on the time to deliver a set of messages. With the new definition comes a new shortcut lemma. The shortcut lemma makes it possible to perform pointer-jumping techniques similar to recursive doubling in a conservative fashion. As a consequence, the minimum-cost spanning forest, connected components, and biconnected components problems can be solved in $O(\lg n)$ steps by conservative algorithms. These algorithms are faster than the corresponding exclusive-read exclusive-write algorithms from the preceding section by a factor of $\lg n$.

A concurrent read or write occurs when two or more processors attempt to read or write the same memory location in a single memory access step. We shall assume that when several processors make requests to read the contents of a location, all of the requests are satisfied. The situation is more complicated when several processors attempt to write to the same location.

We shall assume that there is some simple rule for *combining* multiple write requests to the same location. For example, one of the requests may be arbitrarily chosen to succeed while the others are denied, or the sum of the requests may be written into the location.

2.8.1 A new definition of load

A new measure of load is needed to model the implementation of concurrent reads and writes by an underlying routing network. When several processors request to read a location, it is only necessary for one copy of the data in that location to cross any cut of the underlying network. Similarly, since multiple writes can be combined, at most one message carrying the data to be written into any particular destination needs to cross any cut. The old definition of the load of a set of messages M on a cut S was the number of messages in M whose source is in S and whose destination is in $\overline{S}$, or vice versa. This measure overestimates the number of messages that must cross the cut when some of the messages have the same destination in $\overline{S}$, and can be combined. Consequently, with this measure of load, the load factor is not necessarily a lower bound on the time to deliver a set of messages. The new definition of the load of M on a cut S is the number of different destinations in $\overline{S}$ of messages originating in S, or vice versa. The definitions of a cut, the capacity of a cut, the load factor, and a conservative algorithm remain the same. With the new measure of load, the load factor is a lower bound on the time required to deliver the messages.

The change in the definition of load raises the hope that standard PRAM techniques such as recursive doubling are conservative after all. However, returning to the example of Figure 2-1, we see that after the fourth step, each of the first eight elements in the list points to a *different* element on the other side of the cut. Thus, even with the new definition, the load on the cut has increased from one to eight in three steps. Nevertheless, we will show that a slightly more sophisticated pointer-jumping strategy is conservative.

2.8.2 A shortcut lemma for concurrent reads and writes

The following lemma shows that if all of the pointers into a particular processor are simultaneously shortcut, then the load factor does not increase. Note that unlike the original Shortcut Lemma, the pointer $b \rightarrow c$ is not removed from P.

Lemma 47 *Suppose a data structure consists of a set P of pointers on a set V of vertices and that P contains a pointer $b \rightarrow c$. Let $R = \{x \rightarrow b : x \in V, x \rightarrow b \in P\}$ be the set of pointers in P into b. Then the set Q defined by*

$$Q = P \cup \{x \rightarrow c : x \in V, x \rightarrow b \in R\} - R$$

is conservative with respect to P.

Proof: We will show that $\text{load}(Q,S) \leq \text{load}(P,S)$ for any cut S of the DRAM. There are four ways in which b and c can be assigned to the partition induced by a cut S. Two of the cases can be eliminated by symmetry if we assume that b is on the left side. In both of the remaining cases, the load across the cut is either unchanged or diminished when all of the pointers of the form $x \rightarrow b$ are replaced by pointers $x \rightarrow c$, as shown in Figure 2-6. Note that if b and c lie on the left side of the cut, then all of the pointers into b from the right side of the cut must be shortcut, or the load may increase. □

Corollary 48 *Let B be a set of nodes in V that are independent with respect to P. For each $y \in B$ let $y \rightarrow c(y)$ be a pointer out of y. Let $R = \{x \rightarrow y : x \in V, y \in B, x \rightarrow y \in P\}$ be the set of pointers into the nodes of B. Then the set Q of pointers defined by*

$$Q = P \cup \{x \rightarrow c(y) : x, y \in V, x \rightarrow y \in R\} - R$$

is conservative with respect to P.

Proof: The proof is by induction on the number of nodes in B. □

2.8.3 A conservative pointer jumping technique

The corollary suggests the following conservative tree contraction technique: select a set of independent internal (non-leaf) nodes, then shortcut all of the pointers into those nodes. When the pointers into a node (excluding the root) are shortcut, the node becomes a leaf. Thus, the shortcutting step can reduce, but not increase, the number of internal nodes. The step is repeated until every node in the tree (including the root) points to the root. Such a tree is called a *star*. Note that unlike the tree contraction algorithm from Section 2.5, the number of

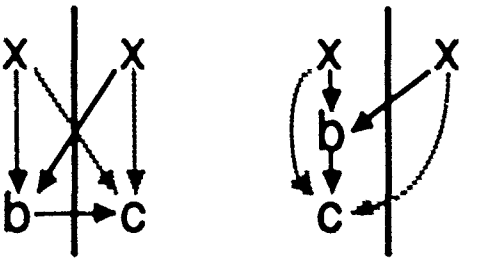

Figure 2-6: A shortcut lemma for concurrent reads and writes. In each of the two cases illustrated, the load factor across the cut is either unchanged or diminished by replacing all of the pointers of the form $x \rightarrow b$ with pointers of the form $x \rightarrow c$.

nodes in the tree does not decrease at each step, and the in-degree of the nodes in the tree can grow.

It is relatively easy to find a large random independent set of internal nodes. First, each internal node chooses to be a candidate with probability 1/2. Next, all candidates whose parents have also initially chosen to be candidates drop out of the running. The remaining candidates form an independent set. At each step, every internal node except the root has probability 1/4 of belonging to the set. Since the root points to itself, it will never be included. By Lemma 42 the probability that at least 1/8 of the internal nodes (excluding the root) belong to the independent set, and thus become leaves, is at least 1/7.

The following lemma shows that if the independent sets are found this way, then the algorithm requires $O(\lg n)$ steps, with high probability.

Lemma 49 *With high probability, the randomized pointer jumping algorithm takes $O(\lg n)$ steps to contract an n-node directed tree to a star.*

Proof: The proof is nearly identical to the third part of the proof of Theorem 44. □

This conservative tree contraction technique can be applied when the input graph has the doubly-linked incidence ring representation from Section 2.7. The representation of a directed tree is itself a binary tree. After applying the tree contraction algorithm to the binary tree, all of the elements in the representation hold pointers to the principal element on the incidence ring of the root. Because an undirected tree can be rooted at any node, and any element on the

incidence ring of the root can be chosen as its principal element, a star rooted at any element in the representation of an undirected tree is conservative with respect to the representation.

2.8.4 A minimum-cost spanning forest algorithm

In this section we present an $O(\lg n)$-step conservative algorithm for finding a minimum-cost spanning forest of an n-node graph. The algorithm is based on the CRCW PRAM minimum-cost spanning forest algorithm of Awerbuch and Shiloach [8], which in turn is based on the connected components algorithm of Shiloach and Vishkin [88].

A minimum-cost spanning forest is defined in Section 2.7. As in that section, we assume without loss of generality that all edge weights are distinct, so that an input graph $G = (V, E)$ has a unique minimum-cost spanning forest F.

The algorithm demarcates the minimum-cost spanning forest by marking edges as belonging to F. Initially no edges are marked. At each step of the algorithm, the currently marked edges form a subforest of F. Each connected component of the subforest is a tree. As in the algorithm from Section 2.7, for each of these components, the algorithm maintains a separate directed tree on the processors in the adjacency-ring representation of the component. However, unlike that algorithm, the edges in the directed tree are not necessarily a subset of the ring and edge pointers. As we shall see, each directed tree is nevertheless conservative with respect to the adjacency-ring input representation of the corresponding component. We denote the set of directed trees $\{T_i\}$, where the index i of the tree is the address of the root. Initially, each node in G is an isolated component, and its directed tree is a star on the nodes of its adjacency ring. When the algorithm terminates, each directed tree is a star on the nodes in the adjacency-ring representation of a different connected component of F.

The algorithm proceeds in phases, each consisting of two basic steps: star-hooking and pointer-jumping. In the star-hooking step, the lowest cost edge connecting each star in $\{T_i\}$ to another component is marked as belonging to F, and the root of the star is made a child of a node in the neighboring component. The pointer-jumping step is the same as that in the tree-contraction algorithm. The algorithm repeats these steps until $\{T_i\}$ consists entirely of stars, and none of these stars have any neighbors in G.

We now describe the star-hooking step in more detail. The first task is to determine which

component (if any) is adjacent to each star via the lowest-cost edge. Each processor in a star whose edge pointer leads outside the star writes the cost of the edge to the root. These concurrent writes are combined using the min operator, so that the lowest edge cost wins. If the star has no neighbors, then it becomes inactive. Also, if two stars select the same edge, then the one with the lower index does nothing. Before the star is hooked into to another tree, it is rerooted at the winning processor. The new root is hooked into the neighboring tree via its edge pointer. If the node at the end of the edge pointer is a leaf, then the edge pointer is shortcut so that the root points to the parent of the leaf. This last operation ensures both that the star-hooking step is conservative and that it does not increase the number of internal nodes in $\{T_i\}$.

The following pair of theorems show that the algorithm is conservative and that it requires $O(\lg n)$ phases, with high probability.

Theorem 50 *With high probability, the algorithm requires $O(\lg n)$ phases to find the minimum-cost spanning forest of an n-node graph.*

Proof: We bound the number of phases using a potential function argument. The quantity of interest is the number of internal nodes in active trees in $\{T_i\}$. Initially, there is a star of height 1 for each of the n nodes in G, so there are n internal nodes. The star-hooking step does not increase the number of internal nodes. After star hooking there are no active stars remaining, so every tree has height at least 2. Since roots are not included in the independent set, in the worst case we expect 1/8 of the internal nodes to be placed in the set. By Lemma 42 the probability that at least 1/16 of the internal nodes become leaves is at least 1/15. The remainder of the proof is like the third part of the proof of Theorem 44. □

Theorem 51 *The algorithm is conservative.*

Proof: The key to the proof is that at the beginning of each phase, the set of directed trees, $\{T_i\}$, is conservative with respect to the adjacency-ring representation of the input graph. The proof is by induction on the number of phases completed. Before the first phase, $\{T_i\}$ consists of a set of stars, one for each node in the input graph. Each star is conservative with respect to the ring pointers in its adjacency ring. Now assume the inductive hypothesis.

The star-hooking step consists of rerooting some stars, and hooking them into adjacent trees. Rerooting is justified because, as we have previously observed, a star rooted at any node in the adjacency-ring representation of the corresponding component is conservative with respect to that representation. When hooking a root into a node in an adjacent tree via an edge pointer, we must be ensure that the edge pointer is shortcut in the same way that any other pointers into that node have been shortcut. If the node is a leaf, then it may have belonged to the independent set in some previous pointer-jumping step. In this case, the root must be hooked into the node's parent. If the node is not a leaf then the pointers into the node have never been shortcut. In this case, the root must be hooked into the node via its edge pointer. In the pointer-jumping step, the pointers into an independent set of the nodes in $\{T_i\}$ are shortcut. By Lemma 48 the resulting set of trees remains conservative with respect to the input representation.

All communication in the algorithm is performed across edge pointers and directed tree pointers. The edge pointers are a subset of the pointers in the input representation, and as we have just proved, the tree pointers are conservative with respect to the representation. □

The algorithm can used as a subroutine in $O(\lg n)$-step algorithms for finding the connected and biconnected components of an n-node graph. The details of the reductions are presented in Section 2.7.

The algorithm can be made deterministic using the deterministic coin-tossing algorithm of Cole and Vishkin [20]. The goal is to find a large independent set of non-root internal nodes without using randomization. Let k be the number of internal nodes. The first step is to remove the leaves of $\{T_i\}$ so that k nodes remain. Next, remove the roots. Since every tree has height at least 1 after the removal of the leaves, at least $k/2$ nodes are left. Next, remove any remaining nodes with 2 or more children. Since there are $k/2$ edges (including the self-pointers at the roots), this step removes at most $k/4$ nodes. At this point the graph consists of chains only of chains. The deterministic coin-tossing technique can be used to select an independent set of at least $k/12$ nodes in $O(\lg^* m)$ steps, where m is the number of processors in the DRAM. Thus, the time for the algorithm is $O(\lg n \lg^* m)$.

2.9 Remarks

This section offers a perspective on the DRAM model. We explore the analogy between PRAM's and universal networks on the one hand, and DRAM's and volume-universal networks on the other. We then discuss the issue of how data structures can be efficiently embedded in DRAM's—a problem not faced in the PRAM model. We also suggest how one might define the load factor for data structures other than graphs, such as matrices. Finally, we offer some comments on how some of our definitions and techniques might be extended or generalized.

The literature contains a large body of results concerning *universal* networks, such as the Boolean hypercube [96]. Universal networks are capable of simulating any PRAM program with at most polylogarithmic degradation in time (see, for example, the simulation [35] of an EREW-PRAM on a butterfly network). In light of this work, one might wonder why the DRAM model should be studied at all.

A potential problem with universal networks is that they may be difficult to physically construct on a large scale. The number of external connections (pins) on a packaging unit (chip, board, rack, cabinet) of an electronic system is typically much smaller than the number of components that the packaging unit contains, and can be made larger only with great cost. When a network is physically constructed, each packaging unit contains a subset of the processors of the network, and thus determines a cut of the network. For a universal network, the capacity of every cut must be nearly as large as the number of processors on the smaller side of the cut; otherwise, the load-factor lower bound would make it impossible to perform certain memory accesses in polylogarithmic time. Thus, when a universal network is physically constructed, the number of pins on a packaging unit must be nearly as large as the number of processors in the unit. Consequently, if all the pin constraints are met, a packaging unit cannot contain as many processors as might otherwise fit. Alternatively, if each packaging unit contains its full complement of processors, then pin limitations preclude the universal network from being assembled.

The impact of pin constraints can be modeled theoretically in the three-dimensional VLSI model [51, 56] where hardware cost is measured by volume and the pinboundedness of a region is measured by its surface area. In this model, the largest universal network that can fit in a given volume V has only about $V^{2/3}$ nodes. In the two-dimensional VLSI model [93], where

pinboundedness is measured by perimeter, the bound is even worse.

Since the density of processors in a physical implementation of a universal network is low, it is natural to wonder whether there are other networks that make more efficient use of hardware. Recently, it has been shown that *fat-trees* [29, 56] are such a class of "volume-universal" networks. A fat-tree of volume V can simulate any other network of comparable volume with only polylogarithmic degradation in time. (Figure 2-7 shows an area-universal fat-tree.) Thus, a fat-tree of volume V can efficiently simulate not only the universal networks with the same volume, but also some networks with almost V nodes. A key component in the proof that fat-trees are volume-universal is an algorithm for routing a set of messages on a fat-tree in time that is at most a polylogarithmic factor larger than the load factor.

With a suitable assignment of capacities to cuts, the DRAM can abstract the essential communication characteristics of volume and area-universal networks without relying in detail on any particular network. Much as the PRAM can be viewed as an abstraction of a hypercube, in that algorithms for a PRAM can be implemented on a hypercube with only polylogarithmic performance degradation, the DRAM can be viewed as an abstraction of a volume or area-universal network. Fast, communication-efficient algorithms on a DRAM with the appropriate cut capacities translate directly to fast, communication-efficient algorithms on, for example, a fat-tree.

We now turn to the problem of embedding data structures in DRAM's, a problem that must be faced by users of conservative algorithms if the algorithms are to run quickly. In general, the problem of determining the best embedding for an arbitrary data structure is NP-complete, but for many common situations, good embeddings can be found. Moreover, there are many situations in which the input graph structure is simple and known *a priori*, and a good embedding may be easy to construct.

To illustrate how the embedding problem can be solved in certain practical situations, consider the class of DRAM's whose cut capacities correspond to area-universal fat-trees. For this class of DRAM's, the recursive structure of the underlying fat-tree network suggests that a divide-and-conquer approach be taken. For example, a subproblem in switch-level simulation of a VLSI circuit is the finding of electrically equivalent portions of the circuit. A naive divide-and-conquer embedding of the circuit on the fat-tree would yield small load factors for every cut.

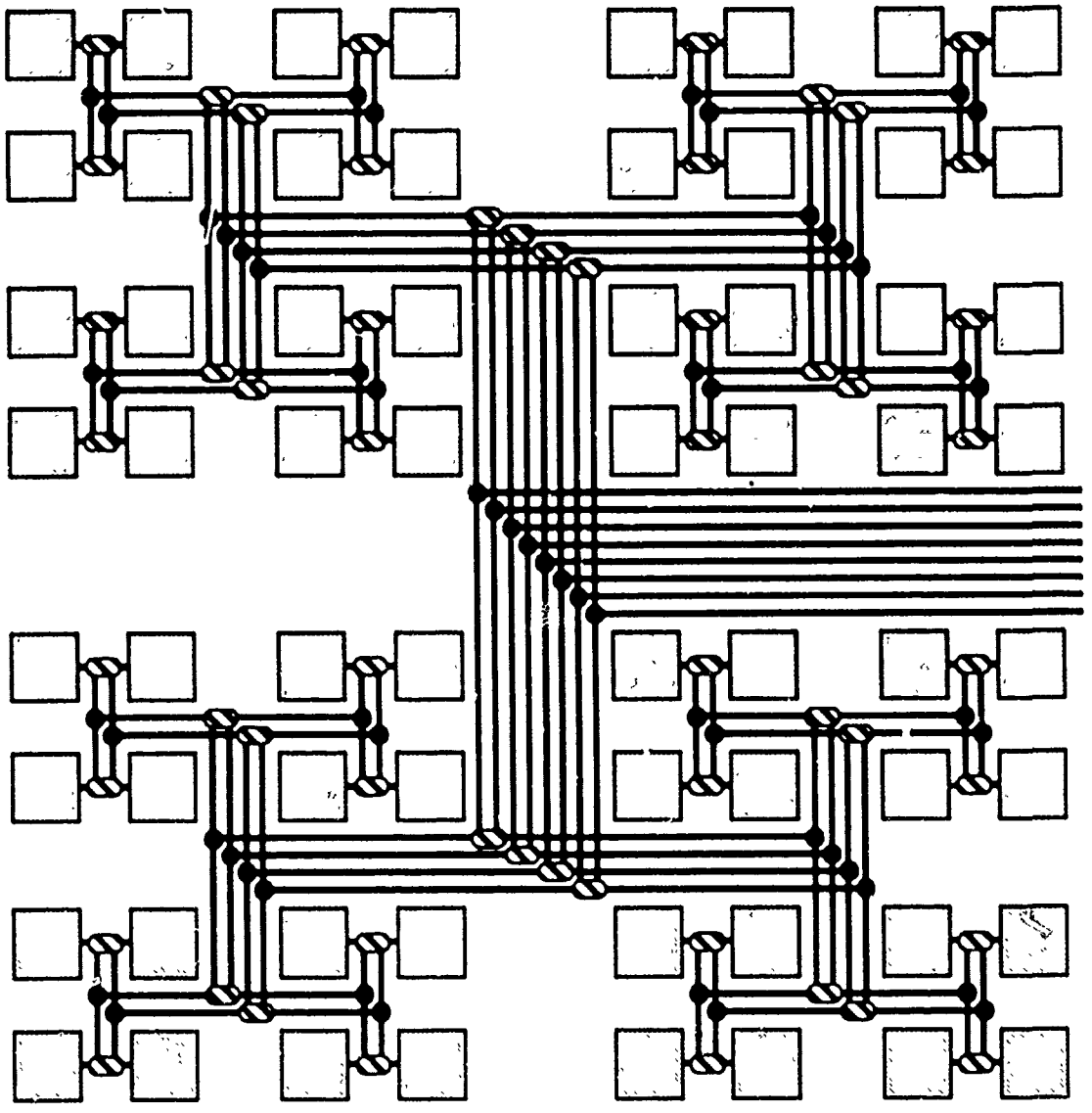

Figure 2-7: A fat-tree network. An area-universal fat-tree, like the one shown, is capable of efficiently simulating any other network of comparable area. Fat-trees are well modeled by distributed random-access machines.

Thus, our conservative connected-components algorithm would never cause undue congestion in communicating messages in the underlying network, and the algorithm would run as fast as on an expensive, high-bandwidth network.

For some graphs, it can be proved that divide and conquer yields near-optimal embeddings on a fat-tree. Specifically, graphs for which a good *separator theorem* [62] exists can be embedded well. Examples include meshes, trees, planar graphs, and multigrids. Situations in which a mesh might be used include systolic array computation [44, 55] and image processing. Planar graphs and multigrids arise from the solution of sparse linear systems of equations based on the finite-element method. Consequently, conservative DRAM algorithms operating on good embeddings of these graphs would run fast on a fat-tree.

The algorithms presented in this chapter operate primarily on graphs for which there is a natural definition of load factor. It is also possible to define the load factor of a data structure that contains no explicit pointers. For example, it is natural to superimpose a mesh on the matrix, as is suitable for systolic array computation [44, 55], and the load factor of the matrix can be defined as the load factor of the mesh.

For some algorithms, the running time may be better characterized as a function of the load factor of the output than the load factor of the input. As an example, consider the problem of sorting a linear list of elements. A natural question to ask is whether a list can be sorted in a polylogarithmic number of steps where at each step, the load factor is bounded by the load factor induced by the linear list together with the permutation determined by the sorted output. Such a sorting algorithm is known for fat-trees [36], but whether such an algorithm exists for general DRAM's is an open question.

Whereas the Shortcut Lemma presented in this chapter holds for any network, for particular networks, other shortcut lemmas may hold. For example, another shortcut lemma for fat-trees is used in [64] to show that an optimal reordering of a linear list in a fat-tree can be determined efficiently by a conservative algorithm on the fat-tree.

As a final comment, we note that the notion of a conservative algorithm may well be too conservative. As a practical matter, it is probably not worth worrying whether every set of memory accesses is conservative with respect to the input, as long as the load factor of memory accesses is not much greater than the input load factor. For example, a contraction tree is not

conservative with respect to its input tree (though the levels of the contraction tree are), but the load factor of the contraction tree is at most $O(\lg n)$ times the input load factor. Algorithms with this looser bound are somewhat easier to code because of the relaxed constraint, and they should perform comparably.

Chapter 3

Work-preserving emulations

3.1 Introduction

In this chapter, we study the problem of emulating an N_G-node *guest* network $G = (V_G, E_G)$ on an N_H-node *host* network $H = (V_H, E_H)$ where $N_H \leq N_G$. Our goal is to emulate T_G steps of any computation on G in $T_H = ST_G$ steps on H where S (the *slowdown* of the emulation) is as small as possible.

The slowdown of the emulation must always be at least as large as N_G/N_H since G has N_G/N_H times as many processors as does H. If $S = O(N_G/N_H)$, then we say that the emulation is *work-preserving* because then the total *work* (i.e., the processor-time product) performed by the emulating network ($W_H = T_H N_H$) is within a constant factor of the work performed by the guest network ($W_G = T_G N_G$). Such emulations achieve optimal speedup (to within a constant factor) over sequential emulations of G since they use N_H processors to solve a problem $\Theta(N_H)$ times faster than is possible with a single processor.

More generally, we say that there is a *work-preserving emulation* of a class of networks $\mathcal{G}$ by a class of networks $\mathcal{H}$ with slowdown $S(N)$ if for every N and T, we can emulate any T steps of any $S(N)N$-node network in $\mathcal{G}$ in $O(S(N)T)$ steps on any N-node network in $\mathcal{H}$. If $S(N) = O(\log^\alpha N)$ for some constant α, then we say that the emulation is *NC work-preserving* since every step of G can be emulated in $O(\log^\alpha N)$ steps of H. If $S(N) = O(N^\alpha)$ for some

This chapter describes joint research with Richard Koch, Tom Leighton, Satish Rao, and Arnold Rosenberg [40].

constant α, then we say that the emulation is *polynomial time work-preserving*, and so on. In the special case that $S(N) = O(1)$, we say that the emulation is *real-time*. Real-time emulations are the hardest to obtain since we require the host network to emulate a guest network of the same size with constant slowdown.

As a simple example, let $\mathcal{G}$ be the class of linear arrays, and $\mathcal{H}$ be the class of all bounded-degree connected networks. It is well known [87] that an N-node linear array can be embedded one-to-one in any connected bounded-degree N-node network with constant dilation and congestion. (By an embedding of a graph G into a graph H, we mean a mapping $\phi : G \rightarrow H$ that maps the nodes of G to the nodes of H and the edges of G to paths in H. The *dilation* of an embedding is the length of the longest path $\phi(e)$ corresponding to an edge of G. The *congestion* of an embedding is the largest number of paths $\phi(e)$ crossing a single edge of H. The *load* of an embedding is the maximum number of nodes of G mapped to a single node of H. In a one-to-one embedding, the load is 1.) Hence any N-node bounded degree connected network H can emulate any N-node linear array with constant slowdown, and thus there is a real-time emulation of the class $\mathcal{G}$ by the class $\mathcal{H}$.

As another simple example, consider the more interesting problem of emulating a butterfly on a linear array. We will prove that the class of butterflies cannot be real-time emulated by the class of linear arrays. (This should come as no surprise, although the proof is not entirely trivial.) However, there is a simple work-preserving emulation of the class of butterflies by the class of linear arrays with slowdown 2^N. In particular, consider an $N2^N$-node butterfly with nodes and edges

$$V = \{\langle i, w\rangle | 1 \leq i \leq N, w \in \{0,1\}^N\}, \text{ and}$$

$$E = \{(\langle i, w\rangle, \langle i', w'\rangle) | i' = i + 1, w' = w \text{ or } w' = w^{(i)}\},$$

where $w^{(i)}$ denotes w except that the ith bit is changed. Then by mapping the 2^N nodes of the form $\langle i, w\rangle$ (where $w \in \{0,1\}^N$) to the ith node of the linear array, an N-node linear array can emulate an $N2^N$-node butterfly with 2^N slowdown.

Seeing this elementary example, one is tempted to ask if there are faster work-preserving emulations of a butterfly on a linear array. In other words, can we emulate a smaller butterfly (say with polynomial blowup) in a work-preserving fashion on a linear array? Although the proof is not obvious, the answer is no. There is no polynomial-time work preserving emulation

of the class of butterflies by the class of linear arrays. Any such emulation requires exponential slowdown. Alternatively, we might wonder if a linear array can emulate any bounded-degree network in a work-preserving fashion given enough slowdown. Again, the answer is no. Although the linear array can emulate a butterfly in a work-preserving fashion, it cannot emulate any expander, no matter how much blowup is allowed. In fact, by combining these results we can conclude that even a butterfly is not sufficiently powerful to emulate an expander in a work-preserving fashion.

We also consider emulations that are not work-preserving. Such emulations are (by definition) inefficient, and we define the inefficiency of such an emulation to be $I = W_H/W_G$. In these terms, an emulation is work-preserving if it has constant inefficiency. Many of our bounds will reflect tradeoffs between slowdown and inefficiency. In general,

$$I = \frac{S}{C}$$

where $C = N_G/N_H$ is the *contraction* of an emulation.

3.1.1 The motivation

There are several good reasons for studying the problem of emulating one network on another in a work-preserving fashion. First, this kind of analysis gives us an excellent means by which to compare the computational power of one network relative to that of another. More importantly, it gives us an automatic way to compile and run algorithms designed for one kind of parallel architecture without loss of efficiency on another. This is provided, of course, that the ratio of the size of the problem to the size of the machine is large enough. For example, we have already seen that a small linear array (which has a very simple structure) is just as efficient in terms of work as a very large butterfly (which has a more complicated structure).

More generally, the study of work-preserving emulations lies at the heart of efficient parallel computing. Indeed, one of the central problems in efficient parallel computing is the task of mapping a collection of processes linked by precedence and/or communication constraints onto the processors and routing network of a parallel machine so that

1. the processing load imposed on the processors is balanced,

2. the communication between processors can be handled efficiently, and

3. the computation and communication can be scheduled so that the necessary inputs for a process are available where and when the process is scheduled to be computed.

In other words, we would like to schedule the communication and computation in a way that takes maximum advantage of the available hardware to minimize the completion time of the job.

In general, we can model the computation to be performed by a DAG. Each node of the DAG represents a process and each directed edge (u, v) represents a communication that must take place between u and v. Typically, this communication represents data output from u after u is completed which is to be input to v before v is started. The parallel machine can be modeled as an undirected network. The nodes of the network correspond to processors, and the edges correspond to communication links between processors (and/or their associated memories). The implementation of the computation to be performed on the parallel machine then corresponds to an embedding of the DAG in the network so that nodes of the DAG are mapped to nodes of the network and so that edges of the DAG are mapped to paths in the network. We may also need to construct a schedule that specifies the communication and computation of the DAG that is being performed during each step of the network. This will be particularly important if the parallel machine is synchronous.

In many applications, the DAG possesses a very natural structure. For example, typical DAGs encountered in practice are derivatives of a binary tree, array, butterfly, or shuffle-exchange graph. This is often due to the fact that the DAG is associated with an algorithm whose inherent underlying structure is a tree or array (as is the case for many problems in numerical analysis and linear algebra) or a butterfly or shuffle-exchange graph (as is the case for Fourier Transform and data manipulation problems). Alternatively, it could be that the DAG was constructed from an algorithm specifically designed for use on one of these common parallel architectures.

Similarly, parallel networks also tend to be very naturally structured and typically are configured as trees, arrays, butterflies, and the like. Hence, the mapping problem often consists of emulating T_G steps of one N_G-node network (represented as a $T_G N_G$-node DAG) on an N_H-node network with a different structure. Ideally, we would like to perform the computation in $O(T_G N_G / N_H)$ steps, which is precisely the problem of finding a work-preserving emulation of

one network on another.

In practice, the guest network can be substantially larger than the host network. For example, it is not uncommon for a parallel machine with between 8 and 256 processors to be emulating array-based computations involving hundreds of thousands of data points. In such examples, even work-preserving emulations with exponential slowdown may be within the scope of practicality. Indeed, the most important feature of the computation is that it be work-preserving.

3.1.2 A closer look at the computational model

If we can find an embedding of a graph G into a graph H with constant dilation, congestion, and load, then it is fairly clear that H can emulate G with constant slowdown. Is the reverse true? Somewhat surprisingly, it is not. For example, Bhatt, Chung, Hong, Leighton and Rosenberg [11] proved that any embedding of an N-node mesh into an N-node butterfly with constant load requires dilation $\Omega(\log N)$, the worst possible. At first glance, it might seem that this result implies that any emulation of an N-node mesh by and N-node butterfly must have slowdown at least $O(\log N)$. However, in this chapter we show that an N-node butterfly can emulate T-steps of an N-node mesh in $O(T \log\log N)$ steps. In [40] we present a more sophisticated emulation scheme that requires only $O(T)$ steps.

In order to understand how such a contradictory result is possible, we need to take a closer look at what it means to emulate T_G steps of one network in T_H steps on another. We start by modeling the computation performed by the guest network G as a pebble DAG Γ. In particular, we will have a pebble for every node-time pair (v,t) where v is a node of G and $0 \leq t \leq T_G$. (Pairs of the form $(v,0)$ correspond to inputs.) In fact, we may have many pebbles associated with a single pair (v,t), which will correspond to the same computation being done more than once. (This is the trick that allows us to emulate a mesh on a butterfly in real time.) To compute any pebble labeled (v,t), we need as inputs pebbles labeled $(v,t-1)$ and $(v_1,t-1),(v_2,t-1),\ldots,(v_k,t-1)$, where $v_1,v_2,\ldots,v_k$ are the neighbors of v in G. We use the directed edges of Γ to denote this dependence in the usual way.

Because many pebbles can have the same label, there are many DAGs Γ associated with any graph G. In order to emulate G on H, we only need to find an embedding and an accompanying

schedule of one of these DAGs in H. Once an embedding and schedule of a DAG is fixed, the emulation proceeds in a standard way. In particular, during each step of the computation, a node of H can

1. make a copy of a single pebble that it contains,
2. send a single pebble to a neighbor, and/or
3. create a pebble with label (v,t) provided that it already contains input pebbles with labels $(v,t-1)$ and $(v_1,t-1),(v_2,t-1),\ldots,(v_k,t-1)$.

Initially, we will allow a node of H to have access to any input, although to use any of these inputs in a meaningful way will take time. By the end of the emulation, we must have computed pebbles with all labels of the form (v,T_G). (For purposes of simplicity, we will use a pebble to denote the state of a processor of G at some particular time, as described above. A more general interpretation would be to use a pebble to denote one of many items (e.g., data and/or functions) stored within a processor. All of our results hold under the more general interpretation, although some of the emulation results become more complicated.)

By allowing several pebbles to have the same label, we dramatically increase the number of possible computation DAGs Γ that correspond to a T_G-step computation of G. This makes it more likely that we can find a computation that can be efficiently emulated on some host network H (e.g., as is the case with emulating a mesh on a butterfly), but it also makes the task of proving lower bounds much more difficult. For example, in order to prove that H cannot emulate G in real-time, we must show that for some T_G, there is no DAG Γ associated with a T_G-step computation of G that can be emulated in $O(T_G)$ steps on H. This can be a formidable task since Γ can look very different than G. Indeed, at the very least, we must choose T_G to be large since by allowing redundant computations of pebbles, any $O(1)$ steps of any N-node bounded-degree graph G can be computed in $O(1)$ steps on any N-node graph H. (This is because if $T = O(1)$, then any output pebble can only depend on $O(1)$ input pebbles, which can be redundantly computed locally since every node of H is assumed to have access to all input pebbles.)

Note that when we prove a lower bound on the ability of a graph H to emulate a graph G, it does not necessarily mean that H cannot effectively compute the same result as does G

(possibly by using a different algorithm, for example). Rather, we are proving lower bounds on the ability of H to perform the same step-by-step computations as G when G is used in a general purpose way. Hence the term *emulation*. We suspect that our pebbling model is probably the most general model in which we could hope to prove lower bounds.

Throughout the chapter we will make use of the fact that if there is an embedding of G in H with congestion c, dilation d, and load l, then there is an emulation of G by H with slowdown $O(l + c + d)$. The follows from the proof in Section 1.2 that for any set of packets whose paths have congestion c and dilation d, there is a schedule of length $O(c + d)$ in which at most one packet traverses each edge at each step. When H is an array, tree, butterfly, or shuffle-exchange graph, the schedule can be computed on-line using the algorithm for layered networks from Section 1.3.

3.1.3 Our results

The technical portion of this chapter is divided into five sections. We commence in Section 3.2 with some general techniques for establishing the existence or nonexistence of a work-preserving emulation. In particular, we describe two general methods for proving lower bounds on the slowdown of a work-preserving emulation. The first method is based on dilation considerations and appears in Section 3.2.1. As an application of this method, we prove that any class of low diameter networks (such as complete binary trees) cannot be emulated in real time on any class of networks that has poor expansion properties (such as arrays of bounded dimension).

The second method is based on congestion properties and is presented in Section 3.2.2. Here we describe a general method for proving that a work-preserving emulation requires a large amount of time, or that it is impossible altogether. As an example, we prove that any work-preserving emulation of a butterfly on an array of bounded-dimension requires exponential time, and that it is not possible to emulate an expander on a butterfly in work-preserving fashion. These results provide a curious contrast between the power of a linear array, butterfly, and an expander. By most standards, it would seem that a butterfly is closer in power to an expander than it is to a linear array. Yet a linear array can emulate a butterfly in a work-preserving fashion, but a butterfly (or most any non-expander) cannot emulate an expander in a work-preserving fashion.

In Sections 3.3 through 3.6, we focus on the special case of emulations by arrays, complete binary trees, butterflies, and shuffle-exchange graphs, respectively. In Section 3.3, we prove tight bounds on the slowdown required for an array to emulate a tree, array or butterfly. In Section 3.4, we prove that there is a work-preserving emulation of bounded-degree trees by complete binary trees with $O(\log\log N)$ slowdown. We also give evidence, but no proof, that there is no corresponding real-time emulation for this class. (Proving that a complete binary tree cannot emulate a complete ternary tree in real-time is one of several challenging questions left open in this chapter.)

In Section 3.5, we show that an N-node butterfly can emulate an N-node mesh with slowdown $O(\log\log N)$. In [40] we show that the emulation can be performed in real-time. This result is interesting because any one-to-one embedding of an array (with dimension 2 or more) in a butterfly requires $\Omega(\log N)$ dilation [11], which suggests that any emulation must require slowdown $\Omega(\log N)$. The result takes on added significance given the fact that many parallel numerical algorithms are array-based while several parallel machines are butterfly-based.

We also describe a simple constant-congestion embedding of an N-node shuffle-exchange graph in an N-node butterfly in Section 3.5. This result has several important consequences. First, it can be used to provide an elementary proof that the N-node shuffle-exchange graph can be laid out in $O(N^2/\log^2 N)$ area and in $O(N^{3/2}/\log^{3/2} N)$ volume. Both results are optimal. The area bound was known previously [38], but the proof was much more difficult (as were the proofs for several suboptimal layouts for the shuffle-exchange graph [34, 48, 50, 90]). The 3-d layout bound is new and was not obtainable by any of the previous approaches to the 2-d layout problem. Second, we apply the result to derive an $O(\log N)$-slowdown work-preserving emulation of the shuffle-exchange graph on the butterfly.

In Section 3.6, we prove the reverse, namely, that there is an $O(\log N)$-slowdown work-preserving emulation of the butterfly on the shuffle-exchange graph. Taken together, these results come very close to resolving a long open question concerning whether or not the butterfly and shuffle-exchange graph are computationally equivalent. In particular, we show that up to NC emulations, the butterfly and shuffle-exchange graphs are equivalent in a work-preserving sense. Thus, for many problems, they can be considered to be computationally equivalent.

As a consequence of the emulations in Section 3.6, we also obtain a real-time emulation

of bounded-degree arrays in the shuffle-exchange graph, and we show how to sort N numbers with high probability in $O(\log N)$ steps on an N-node shuffle-exchange graph. Although the proof of the sorting bound is elementary, it resolves an open question concerning the difficulty of randomized sorting algorithms on the shuffle-exchange graph. Previously, such an algorithm was known for the butterfly [53, 76, 84] but that algorithm made crucial use of the recursive structure of the butterfly, a structure not present in a shuffle-exchange graph.

3.1.4 Previous work

There has been a great deal of previous work on graph embeddings with the intent of showing that one network can or can't emulate another network efficiently [11, 12, 13, 28, 53, 80]. Many of the results were positive and proved things like "all N-node binary trees can be emulated in constant time on an N-node hypercube." There were also some negative results, but because of the lack of a good model, their significance is now less clear. For example, even though an embedding of a mesh into a butterfly requires dilation $\Omega(\log N)$, we now find that a butterfly can emulate a mesh with constant slowdown.

The notion of work-preserving emulations in PRAM models has previously been studied [42, 67] and served to motivate this work. Related problems of scheduling computations on fixed-connection networks have also been studied [72].

3.2 Lower bounds

In this section we present lower bounds on slowdown and inefficiency. Loosely speaking, these lower bounds apply when the guest graph expands faster than the host graph. The first lower bound can be used to show that any emulation of a complete binary tree by a linear array has slowdown $\Omega(N_H/\log N_H)$. The second can be used to show that a butterfly cannot perform a work-preserving emulation of an expander graph, that any work-preserving emulation of a butterfly by a linear array H requires slowdown at least $2^{\Omega(N_H)}$, and that any work-preserving emulation of a $k+1$-dimensional mesh by a k-dimensional mesh H requires slowdown at least $\Omega(N_H^{1/k})$. All of these lower bounds on slowdown are tight.

Before proving the lower bounds, we need to introduce some notation. For an undirected

graph $G = (V, E)$, let $\delta(u, v)$ be the length (number of edges) of the shortest path between nodes u and v in G. Let $B_G(u, i) = \{v \in V | \delta(u, v) \leq i\}$ be the set of nodes within a distance i of u in G and let $b_G(u, i) = |B_G(u, i)|$. We call b_G the growth function of G.

3.2.1 Distance-based lower bound

The following theorem shows that if the guest graph grows faster than the host graph, then any emulation of the guest by the host must be slow.

Theorem 52 *Let $H = (V_H, E_H)$ be an N_H-node host graph and $G = (V_G, E_G)$ be an N_G-node guest graph, and suppose that there are integers τ_H and τ_G such that*

$$\max_{u \in V_H} \sum_{i=1}^{\tau_H} b_H(u, i) < \min_{v \in V_G} \sum_{j=1}^{\tau_G} b_G(v, j).$$

Then any emulation of $T_G \geq \tau_G$ steps of G by H has slowdown

$$S > (\tau_H + 1)/2\tau_G.$$

Proof: The basic idea is to find a sequence of T_G/τ_G pebbles in any T_G-step pebble DAG of G such that each pair of pebbles is separated by at most τ_G guest time steps but are created in H at least τ_H host time steps apart. As we shall see, such a sequence exists only if the slowdown $S = T_H/T_G$ is at least $(\tau_H + 1)/2\tau_G$.

We start the sequence with the last pebble created by H. Suppose that at time T_H some node $u_0 \in V_H$ creates a pebble for DAG node (v_0, t_0), where $t_0 = T_G$. The pebble for (v_0, t_0) cannot be created by H until pebbles for all of its predecessors in the DAG are created. In particular, there are at least $\sum_{j=1}^{\tau_G} b_G(v_0, j)$ precedessors for time steps $t_0 - \tau_G$ through $t_0 - 1$. We want to show that the pebble for at least one of these predecessors must have been created by the host graph before time $T_H - \tau_H$. The pebble for every predecessor of (v_0, t_0) that is created at distance i from u_0 in H must be created at or before time $T_H - i$. Thus at most $\sum_{i=1}^{\tau_H} b_H(u_0, i)$ pebbles for predecessors of (v_0, t_0) are created by H between time steps $T_H - \tau_H$ and $T_H - 1$. Since $\max_{u \in V_H} \sum_{i=1}^{\tau_H} b_H(u, i) < \min_{v \in V_G} \sum_{j=1}^{\tau_G} b_G(v, j)$, the pebble for some predecessor (v_1, t_1), $t_1 \geq T_G - \tau_G$, must be created by the host graph at or before time $T_H - (\tau_H + 1)$.

We can repeat the argument to find a pebble for a predecessor (v_2, t_2), $t_2 \geq T_G - 2\tau_G$, of (v_1, t_1) that must be created by the host at or before time $T_H - 2(\tau_H + 1)$, and so on. Eventually we obtain a pebble (v_k, t_k) such that $\tau_G > t_k \geq T_G - k\tau_G$. This pebble must be created by the host at or before time $T_H - k(\tau_H + 1)$. We assume that input pebbles are created at host time step 0, and that the emulation begins with time step 1. Thus, $T_H - k(\tau_H + 1) \geq 0$. Combining these inequalities, we have

$$T_H/T_G > (\tau_H + 1)/2\tau_G$$

for $T_G \geq \tau_G$. □

Corollary 53 *Any such emulation has inefficiency*

$$I > \Omega\left(\frac{\tau_H N_H}{\tau_G N_G}\right).$$

Corollary 54 *Any emulation of a complete binary tree, G, by a k-dimensional mesh, H, has slowdown at least* $\Omega\left((N_G/\log^k N_G)^{1/(k+1)}\right)$.

Proof: Apply Theorem 52 with $\tau_G = \Theta(\log N_G)$, and $\tau_H = \Theta\left((N_G \log N_G)^{1/(k+1)}\right)$. □

3.2.2 Congestion-based lower bound

The second lower bound requires a little more notation. Let $G = (V, E)$ be an undirected graph as before. For a set $U \subseteq V$, we define the i-neighborhood of U to be the set of nodes within a distance i of some node in U, $\mathcal{N}_i(U) = \cup_{u \in U} B_G(u, i) - U$. We define an $(R, f(R))$-decomposition of G to be a partition of V into $|V|/R$ sets of nodes (regions) such that each contains R nodes and has a 1-neighborhood of size at most $f(R)$.

The last graph parameter that we need, z_G, is best described in terms of a simple game. The player starts by choosing a nodes of a connected graph G and placing them in a bag. The player is given a collection of εa, $0 \leq \varepsilon < 1$, tokens to play with. The game is played in rounds, each consisting of two steps. In the first step, all of the neighbors of the nodes in the bag are added to the bag. In the second step, the player may exchange tokens for nodes in the bag on a one-for-one basis. Let X_i be the set of nodes in the bag at the end of round i, and let Y_i be the set of nodes removed in the second step of round i. Then X_i is given by the recurrence $X_i = X_{i-1} + \mathcal{N}_1(X_{i-1}) - Y_i$. The game ends when the number of nodes in the bag exceeds

it capacity, c, at the end of a step, where $c < N_G$. If k is the number of rounds played, then $|X_i| \leq c$ for $i < k$, $|X_i| > c$ for $i = k$, and $\sum_{i=1}^{k} |Y_i| \leq \varepsilon a$. The goal is to play as many rounds as possible. Let $z_G(a, \varepsilon, c)$ be an upper bound that is non-increasing in a on the length of the longest possible game.

Theorem 55 *Suppose that $H = (V_H, E_H)$ is an N_H-node host graph with an $(R, f(R))$-decomposition, and that $G = (V_G, E_G)$ is an N_G-node guest graph. Let*

$$\beta = \max\left\{z_G\left(\frac{N_G}{4}, 0, \frac{3N_G}{4}\right), z_G\left(\frac{3N_G R}{8N_H}, \frac{1}{2}, \frac{N_G}{2}\right)\right\}.$$

Then for any emulation of G by H where $T_G > 3\beta$,

$$I \geq \min\left\{\frac{R}{32\beta f(R)}, \frac{N_H}{96R}\right\}.$$

Proof: The basic strategy is to show that either the host spends a lot of time passing pebbles across the perimeters of the regions in the $(R, f(R))$-decomposition, or the host spends a lot of time creating pebbles. We will break the T_G guest time steps into blocks of 3β consecutive steps and classify every block as either an *importer* or a *creator*. If a block is an importer, then many pebbles for the block cross region perimeters. If a block is a creator, then some region creates many pebbles for the block. If the majority of the blocks are importers, then the time required by the host to pass pebbles across the perimeters of the regions large. Otherwise, the time required to create the pebbles is large.

Before we can get started we need one more piece of notation. For each node v in G there is at least one pebble created by H for each guest time step t between 1 and T_G. The first pebble created for v for time t is called the *t-primary pebble* for v. For each value of t there are exactly N_G t-primary pebbles. The t-primary pebbles are ordered according to the order in which they are created by H, with ties broken arbitrarily. We call the first $3N_G/4$ t-primary pebbles the *t-early* pebbles and the last $3N_G/4$ the *t-late* pebbles.

We begin with the definition an importer block. Consider a block from step t to $t - 3\beta + 1$. The average number of t-early pebbles created by each of the N_H/R regions in the decomposition of H is at least $p = 3N_G R/4N_H$. We say that a region is *t-busy* if it creates at least $p/2$ t-early

pebbles. We say that a t-early pebble is t-busy if it is created by a t-busy region. At least half of the t-early pebbles are t-busy. Thus, there are at least $3N_G/8$ t-busy pebbles. Suppose that a t-busy region creates $s \geq p/2$ t-busy pebbles. We say that the region is an *importer* if it imports at least $s/2$ pebbles for time steps between $t-1$ and $t-2\beta$. We say that a block is an importer if every t-busy region is an importer, or if some region imports at least $3N_G/16$ pebbles for time steps between $t-1$ and $t-2\beta$. In a importer block, a total of at least $3N_G/16$ pebbles for time steps between $t-1$ and $t-2\beta$ are imported by all of the regions.

If at least half of the $T_G/3\beta$ blocks are importers, then we can find a lower bound on inefficiency by computing the time required to import pebbles. In this case, the total number of pebbles imported by all of the importer blocks is at least $T_G N_G/32\beta$. The host time required to import these pebbles is at least $T_H \geq T_G N_G R/32\beta N_H f(R)$, because at each host time step, each of the N_H/R regions can import at most $f(R)$ pebbles. In this case,

$$I \geq R/32\beta f(R).$$

As we shall see, if a block is not an importer then some region must create many pebbles for the block. Hence the name creator. In a creator block there must be some t-busy region $\mathcal{R}$ that creates $s \geq p/2$ t-busy pebbles but imports fewer than $s/2$ pebbles for time steps between $t-1$ and $t-2\beta$. The t-busy pebbles created by $\mathcal{R}$ cannot be created until pebbles for all of their predecessors in the pebble DAG are created. Since $z_G(s, 1/2, N_G/2) \leq z_G(p/2, 1/2, N_G/2) \leq \beta$, $\mathcal{R}$ imports at most $s/2$ pebbles for time steps between $t-1$ and $t - z_G(s, 1/2, N_G/2)$. Thus $\mathcal{R}$ must create at least $N_G/2$ pebbles for time step $t - z_G(s, 1/2, N_G/2)$. Furthermore, since $\mathcal{R}$ imports at most $3N_G/16$ pebbles for time steps between $t-1$ and $t-2\beta$, it must create at least $5N_G/16$ pebbles for every time step between $t - z_G(s, 1/2, N_G/2)$ and $t-2\beta$. For each of these time steps, at least $N_G/16$ of the pebbles are created for nodes whose $(t-2\beta)$-primary pebbles are $(t-2\beta)$-late pebbles. We call these pebbles the *descendant* pebbles.

We have chosen the descendant pebbles so that none are created by H until all of the descendant pebbles for previous blocks have been created. The early pebbles for all time steps at or before $t-2\beta-z_G(N_G/4, 0, 3N_G/4)$ must be created before the $(t-2\beta)$-late pebbles because $3N_G/4$ nodes in G lie within a distance $z_G(N_G/4, 0, 3N_G/4)$ of the nodes corresponding to the first $N_G/4$ $(t-2\beta)$-primary pebbles. Since $z_G(N_G/4, 0, 3N_G/4) \leq \beta$, the early pebbles for previous blocks must be created before the $(t-2\beta)$-late pebbles. Furthermore, the $(t-2\beta)$-late

pebbles must be created before the descendant pebbles, which in turn must be created before the t-busy pebbles for $\mathcal{R}$.

If at least half of the blocks are creators, then we can derive a lower bound on inefficiency by summing the time to create the descendant pebbles for each of the creator blocks. For each of $T_G/6\beta$ creator blocks, at least $\beta N_G/16$ descendant pebbles are created by a single region. The host time for each block is at least $\beta N_G/16R$. The host time for all of the creator blocks is at least $T_G N_G/96R$ and the inefficiency is at least

$$I \geq N_H/96R.$$

Combining the two cases proves the theorem. □

Corollary 56 *A k-dimensional mesh H cannot perform a work-preserving emulation of an expander graph G.*

Proof: Apply Theorem 55 with $R = \Theta((N_H \log N_H)^{k/(k+1)})$, $f(R) = O(R^{(k-1)/k})$, and $\beta = O(\log(N_H/R))$. The inefficiency is at least $I \geq \Omega((N_H/\log^k N_H)^{1/(k+1)})$. □

Corollary 57 *A butterfly network H cannot perform a work-preserving emulation of an expander graph G.*

Proof: Apply Theorem 55 with $R = \Theta(N_H \log\log N_H/\log N_H)$, $f(R) = O(R/\log R)$, and $\beta = O(\log(N_H/R))$. The inefficiency is at least $I \geq \Omega(\log N_H/\log\log N_H)$. □

Corollary 58 *Any work-preserving emulation of a butterfly G by a k-dimensional mesh H has slowdown at least $2^{\Omega(N_H^{1/k})}$.*

Proof: Apply Theorem 55 with $R = \Theta((N_H \log N_G)^{k/(k+1)})$, $f(R) = O(R^{(k-1)/k})$, and $\beta = O(\log N_G)$. The inefficiency is at least $I \geq \Omega((N_H/\log^k N_G)^{1/(k+1)})$. □

Corollary 59 *Any work-preserving emulation of a j-dimensional mesh G by a k-dimensional mesh H, $j > k$, has slowdown at least $\Omega(N_H^{(j-k)/k})$.*

Proof: Apply Theorem 55 with $R = \Theta((N_G^{1/j} N_H)^{k/(k+1)})$, $f(R) = O(R^{(k-1)/k})$, and $\beta = O(N_G^{1/j})$. The inefficiency is at least $I \geq \Omega((N_H^j/N_G^k)^{1/j(k+1)})$. □

3.3 Emulations by arrays

Although the arrays cannot perform real-time emulations of graphs with small diameter, we can show that they can perform work-preserving emulations of complete binary trees, other arrays, and butterflies. In each case, we find an embedding of the guest graph into the array with acceptable load, congestion, and dilation. The edges of the guest graph are emulated by routing packets between the nodes of the linear array. All of the following results can be shown to be tight by Corollaries 54, 58, and 59.

Observation 60 *An N-node k-dimensional mesh can perform a work-preserving emulation of an $N^{(k+1)/k}/\log N$-node complete binary tree.*

Proof: An $N^{(k+1)/k}/\log N$-node complete binary tree can be embedded in an N-node k-dimensional mesh with load $O(N^{1/k}/\log N)$, dilation $O(N^{1/k}/\log N)$, and congestion $O(N^{1/(k+1)})$. □

Observation 61 *An N-node k-dimensional mesh can perform a work-preserving emulation of an $N^{j/k}$-node j-dimensional mesh, $j > k$.*

Proof: An $N^{j/k}$-node j-dimensional mesh can be embedded in an N-node k-dimensional mesh with load $N^{(j-k)/k}$, congestion $N^{(j-k)/k}$, and dilation 1. □

Observation 62 *An $N_H = n^k$-node k-dimensional mesh H can perform a work-preserving emulation of an $N_G = n2^n$-node butterfly graph G.*

Proof: An $n2^n$-node butterfly graph with 2^n rows and n columns can be embedded in a $N_H = n^k$-node k-dimensional mesh with load $O(2^n/n^{k-1})$, congestion $O(2^n/n^{k-1})$, and dilation $O(n)$. □

It is interesting to note that every connected network can perform a real-time emulation of a linear array. Hence, Observations 60 through 62 can be modified to hold for all connected networks.

3.4 Emulations by complete binary trees

3.4.1 Work-preserving emulations of bounded-degree trees

In this section, we show that any $N \log\log N$-node forest with maximum degree Δ can be embedded in an N-node complete binary tree with load $O(\Delta \log\log N)$, congestion $O(\Delta^2 \log\log N)$, and dilation $O(\log \Delta)$. As a corollary, there is a work-preserving emulation with slowdown $O(\log\log N)$ of the class of bounded-degree forests by the class of complete-binary trees.

In constructing the embedding, we use the following well-known weighted-separator lemma and its corollaries.

Lemma 63 *Suppose that $F = (V, E)$ is a forest where each vertex has been assigned some non-negative weight. Then it is possible to remove a single vertex from V so that the remaining vertices can be partitioned into two subforests F_1 and F_2 such that no edge connects a vertex in F_1 with a vertex in F_2, and F_1 and F_2 each contain at most 2/3 of the total weight.*

Proof: Omitted.

Corollary 64 *By removing a single vertex, it is possible to partition a forest $F = (V, E)$ into two subforests each containing at most $2|V|/3$ vertices.*

Proof: Assign each vertex weight 1 and apply Lemma 63. □

Corollary 65 *By removing a set S of k vertices, it is possible to partition a forest $F = (V, E)$ into two subforests, F_1 and F_2, each containing at most $|V|(1 + (2/3)^k)/2$ vertices.*

Proof: Initially F_1 and F_2 are empty and a third set R contains all of the vertices. Iterate the following step k times. Apply Corollary 64 to split R into two subforests, then remove the smaller subforest from R and add it to the smaller of F_1 and F_2. At the end of each step, F_1 and F_2 differ in size by at most $|R|$. After k iterations, R contains at most $|V|(2/3)^k$ vertices. Add R to the smaller of the two sets. □

Corollary 66 *Suppose that $F = (V, E)$ is a forest where each vertex has been assigned some non-negative weight. Then it is possible remove a set S of k vertices such from V such that the*

remaining vertices can be partitioned into two subforests F_1 and F_2 such that no edge connects a vertex in F_1 with a vertex in F_2, and each contains at most $|V|(1+(2/3)^{\lfloor (k-1)/2 \rfloor})/2$ vertices and at most 5/6 of the total weight.

Proof: First apply Lemma 63 to partition the forest into two subforests L and R, each containing at most 2/3 of the weight. Next, apply Corollary 65 to split L into L_1 and L_2, and R into R_1 and R_2. Let L_1 and R_1 have more weight than L_2 and R_2 respectively. Then both L_1 and R_1 have at most 2/3 of the weight, and L_2 and R_2 have at most 1/6. Let $F_1 = L_1 \cup R_2$ and $F_2 = L_2 \cup R_1$. □

With these tools in hand, we present the embedding.

Theorem 67 *An $N \log\log N$-node forest with maximum degree Δ can be embedded in an N-node complete binary tree with load $l = O(\Delta \log\log N)$, congestion $c = O(\Delta^2 \log\log N)$, and dilation $d = O(\log \Delta)$.*

Proof: The embedding begins by using Corollary 66 to find a set S of $k \in O(\log\log N)$ nodes that partitions the forest $F = (V, E)$ into two subforests, each containing at most $|V|(1 + 1/\log N)/2$ vertices. We embed S at the root of the binary tree and then recursively embed one of the subforests in the left subtree of the root, and the other in the right.

At levels below the root, we use Corollary 66 to simultaneously partition the vertices of the forest and the edges connecting the forest to vertices that are embedded higher in the binary tree. Let $F_i = (V_i, E_i)$ be a forest to be embedded in a subtree rooted at a level i node v_i in the binary tree. Let N_i be the number of edges connecting F_i to vertices embedded higher in the binary tree; N_i is the congestion of the binary tree edge connecting v_i to its parent. We assign each vertex of F_i a weight equal to the number of neighbors it has that are embedded higher in the binary tree. Using Corollary 66, we find a set S_i of k vertices that partitions F_i into two subforests, each of size at most $|V_i|(1 + 1/\log N)/2$, and each having at most $(5/6)N$ edges to vertices that are embedded higher in the tree. We embed the vertices of S_i at v_i and recursively embed one of the subforests in the left subtree of v_i, and the other in the right subtree.

To limit the dilation to some integer d, whenever i is a multiple of d we embed at v_i not only S_i but also all of the vertices in F_i that have at least one neighbor embedded somewhere higher in the binary tree.

We must now show how to choose d so that both the congestion and the load of the embedding are small. Consider any simple path from a level i node v_i in the binary tree to a level $i+d$ node, v_{i+d}, where i is a multiple of d. At level i, we embed a separator of size k and at most N_i other vertices that have at least one neighbor embedded higher in the tree. Since each of these vertices has at most Δ neighbors, $N_{i+1} \leq \Delta k + \Delta N_i$. At level $i+1$, we embed a separator of size k that partitions F_{i+1} into two subforests, each having at most $(5/6)N_{i+1}$ edges to vertices embedded higher in the binary tree. Thus, at level $i+2$, we have $N_{i+2} \leq (5/6)N_{i+1} + \Delta k$. In general, N_{i+j} is given by the recurrence

$$N_{i+j} \leq \begin{cases} \Delta k + \Delta N_i & j = 1 \\ (5/6)N_{i+j-1} + \Delta k & 1 < j \leq d \end{cases}$$

Solving the recurrence yields

$$N_{i+j} \leq 6\Delta k + (5/6)^{j-1}\Delta N_i.$$

We are now in a position to calculate the load and the congestion. The preceding argument shows that for $d \in O(\log \Delta)$ and $N_i \in O(\Delta k)$, we have $N_{i+d} \leq N_i$. Thus, in every simple path between a node at level i and a node at level $i+d$, where i is a multiple of Δ, the congestion starts at $O(\Delta k)$ at level i, rises to at most $O(\Delta^2 k)$ at level $i+1$ and proceeds to drop back down to at most $O(\Delta k)$ at level $i+d$. Thus, the congestion of the embedding is at most $O(\Delta^2 \log\log N)$. How large can the load be? At each node of the binary tree we embed a separator of size k. For every i that is a multiple of d, we also embed a set nodes of size $N_i = O(\Delta k)$. Finally, at the leaves we embed forests of size

$$N \log\log N((1 + 1/\log N)/2)^{\log N},$$

which is at most $O(\log\log N)$. Thus the load is at most $O(\Delta \log\log N)$. □

Corollary 68 *There is a work-preserving emulation of the class of bounded-degree forests by the class of complete-binary trees with slowdown* $O(\log\log N)$.

3.4.2 Congestion lower bound for complete ternary trees

In this section we show that any embedding of an N-leaf complete ternary tree T_3 in an M-leaf complete binary tree T_2, $N < M < 3N$, in which the leaves of T_3 are mapped to the leaves of

T_2 with load at most $2^{\log^\alpha N}$, fixed $\alpha < 1$, has congestion at least $\Omega(\sqrt{\log \log N})$. This lower bound suggests, but does not prove, that real-time emulation of a complete ternary tree by a complete binary tree is impossible.

Theorem 69 *Any embedding of an N-leaf complete ternary tree T_3 in an M-leaf complete binary tree T_2, $N < M < 3N$, in which the leaves of T_3 are mapped to the leaves of T_2 with load $l = 2^{\log^\alpha N}$, fixed $\alpha < 1$, has congestion at least $\Omega(\sqrt{\log \log N})$.*

Proof: The proof has the following outline. Let L denote the number of leaves of T_3 in a subset S of the nodes of T_3, and let w be a base-3 string representing L. First we show that for any S, the number of 1's in w is at most one plus the number of edges between S and $\overline{S}$. As a consequence, if S is the set of nodes mapped to a subtree rooted at a node v in T_2, then the congestion on the edge from the v to its parent is at least as large as the number of 1's in w. Next, we construct a path $v_0, v_1, \ldots, v_{\log M}$ in T_2 from the root v_0 to a leaf $v_{\log M}$ such that there is a long sequence of nodes on the path, $v_j, v_{j+1}, \ldots, v_{j+s-1}$ such that for each v_i, where $j \leq i \leq j+s-1$, the number of leaves of T_3 mapped to the left and right subtrees of v_i are nearly equal. Let S_i be the set of nodes of T_3 mapped to the subtree rooted at v_i, let L_i be the number of leaves of T_3 in S_i, and let w_i be the base-3 string representing L_i. To complete the proof we show that for some i, where $j \leq i \leq j+s-1$, there are many 1's in w_i.

First we show that for any subset S of the nodes of T_3, the number of 1's in w is at most $|E_S| + 1$, where E_S is the set of edges in T_3 connecting a node in S to a node in $\overline{S}$. The key idea is that L can be expressed as a series of $|E_S| + 1$ terms, both positive and negative, where each term is a perfect power of 3. If the root of T_3 belongs to S, then the series begins with the term N; otherwise it begins with 0. Thereafter, each edge in E_S contributes a term to the series. An edge between a node u on level l and its parent on level $l-1$ contributes $N/3^l$ if u is in S, and $-N/3^l$ otherwise. Because adding or subtracting a power of 3 can produce at most one 1 digit in a base-3 number, the number of 1's w is at most $|E(S)| + 1$.

Starting at the root, v_0, we construct the path in T_2 according to the following rule. Suppose that v_i is a node on the path. Then the next node on the path, v_{i+1} is the root of the left or right subtree of v_i containing more leaves of T_3. Let L_i be the number of leaves of T_3 mapped to the subtree rooted at v_i. Then v_{i+1} contains at least $L_i/2$ leaves of T_3. We call the split at

v_i *fair* if both of its subtrees contain at most $L_i(1/2+\varepsilon)$ leaves of T_3, where ε will be specified later.

Next we put a lower bound on the length of the longest sequence of consecutive fair splits. Let b be the number of unfair splits on the path. The number of leaves of T_3 mapped to the leaf at the end of the path is at least

$$\left(\frac{1}{2}\right)^{\log M-b}\left(\frac{1}{2}+\varepsilon\right)^{b}.$$

Since the load is at most l, and $1+x \le e^{x/2}$ for $0 \le x \le 1$, we have $l \ge \frac{1}{3}e^{\varepsilon b}$. Let s be the length of the longest sequence of consecutive fair splits. Then $s \ge \log M/b \ge \varepsilon \log M/\ln 3l$.

We now show that in the longest sequence of consecutive fair splits $v_j, v_{j+1}, \ldots, v_{j+s-1}$, there must be a node v_i, where $1 \le i \le j+s-1$ such that there are many 1's in w_i. For the moment, let us assume that at each node v_i on the sequence, the number of leaves of T_3 mapped to each subtree of v_i is exactly $L_i/2$. Then we can prove that at some node v_i on the sequence, the number of 1's in the t most significant digits of w_i is at least $\sqrt{t}$, where $t = (\log_3 2)s$. Suppose that the the number of 1's in w_j is smaller than $\sqrt{t}$ (otherwise we're done). The 1's in w_j partition it into substrings consisting of 0's and 2's only. In each substring, division by 2 either converts all of the 0's to 1's (leaving the 2's unchanged), or converts all of the 2's to 1's (leaving the 0's unchanged). Thus, after division by 2, 0's and 2's are adjacent in at most $\sqrt{t}$ places in w_{j+1}. Thus, there must be a substring of either $\sqrt{t}$ 0's or $\sqrt{t}$ 2's in w_{j+1}. In either case, after at most s divisions by 2 the substring is converted to all 1's.

Unfortunately, a fair split at a node v_i does not divide L_i exactly by 2; it also adds as much as εL_i. For $\varepsilon \le 1/3^t$, adding εL_i does not change the t most significant bits unless a carry propagates in. We need to show that our substring of $\sqrt{t}$ 0's or 2's is not adversely affected by carries. Since a carry into a substring of 2's turns them all into 0's, we need only consider the effect of a carry into a substring of 0's. A carry into a substring of 0's converts the least significant 0 in the substring into a 1, which is bad, because it reduces the length of the string. However, $3^{\sqrt{t}/2}$ carries are required to modify the $\sqrt{t}/2$ least significant 0's in the substring. Since at most one carry occurs at each of the s splits, and $s \ll 3^{\sqrt{t}/2}$, the length of the longest string of 0's never drops below $\sqrt{t}/2$.

To finish, we choose values for ε, l, and t. To make the lower bound strong, we want to make t large without making l too small. For any fixed $\alpha < 1$, we can choose $l = 2^{\log^\alpha N}$,

$t = \Theta(\log\log N)$, and $\varepsilon = 1/3^t$. The congestion is at least $\sqrt{t}/2 = \Omega(\sqrt{\log\log N})$. □

3.5 Emulations by butterfly networks

3.5.1 Work-preserving emulations of binary trees

When the Bhatt, Chung, Hong, Leighton, Rosenberg result [11] that a butterfly can emulate a complete binary tree in real-time is combined with the material in Section 3.4, we find that there is an $O(\log\log N)$-time work-preserving simulation of the class of binary trees on the butterfly. Whether or not this emulation can be performed in real-time remains an open question.

3.5.2 Emulation of meshes

In this section we show that an $O(N)$-node butterfly can emulate an N-node mesh with slowdown $O(\log\log N)$.

Theorem 70 *An $O(N)$-node butterfly can emulate T steps of a $\sqrt{N} \times \sqrt{N}$ mesh in $O(T\log\log N)$ steps.*

Proof: The trick is to divide the mesh into slightly overlapping submeshes, as shown in Figure 3-1. Each $\log^2 N \times \log^2 N$ submesh overlaps its neighbors in either $2\log N$ rows or $2\log N$ columns. Since the submeshes overlap, some mesh nodes appear in as many as four submeshes. We call two nodes in neighboring submeshes *mates* if they correspond to the same mesh node. Each submesh is emulated by a different $O(\log^4 N)$-node subbutterfly. Since a single mesh node may be emulated by several subbutterflies, the butterfly performs redundant computation.

A subbutterfly emulates the corresponding submesh by routing packets between each mesh node and its neighbors. Since, an $O(\log^4 N)$-node subbutterfly can route any permutation of $O(\log^4 N)$ packets in $O(\log\log N)$ steps, the time to emulate each step of the submesh is $O(\log\log N)$.

The nodes on the borders of a submesh cannot be emulated by the corresponding subbutterfly because they require inputs from mesh neighbors that the subbutterfly does not emulate. As a consequence, nodes at distance δ from the border can be emulated for only δ steps. Fortunately, every node at distance $\delta \leq \log N$ from the border of one submesh has a mate at

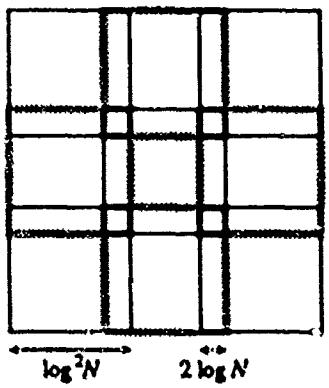

Figure 3-1: The division of the mesh into submeshes. Each $\log^2 N \times \log^2 N$ submesh overlaps its neighbors in either $2\log N$ rows or $2\log$ columns.

a distance of $2\log N - \delta \geq \log N$ in a neighboring submesh. Thus, every mesh node can be emulated for the full $\log N$ steps in some subbutterfly.

To emulate $T > \log N$ steps of the mesh, the T steps are broken into *blocks* of $\log N$ consecutive steps. The time to emulate a block of $\log N$ steps is $O(\log N \log\log N)$. Before the next block can be emulated, the nodes within distance $\log N$ of the borders of the submeshes must be updated by their mates. Since an N-node butterfly can route any permutation of N-packets in $O(\log N)$ steps, the updating takes $O(\log N)$ time. The total time for T steps is $O(T\log\log N)$. □

This emulation scheme has two main drawbacks. First, the packets that are sent between blocks to update the mates must each carry enough information to reflect the change in the state of a mesh node over a period of $\log N$ steps. Such packets are unreasonably large. This problem can be overcome by observing that only $O(N/\log N)$ of the mesh nodes must be updated. If these nodes are carefully positioned within their subbutterflies, it is possible to route $\log N$ packets to each of them in $O(\log N)$ steps. Second, the slowdown is too large. The slowdown can be reduced from $O(\log\log N)$ to $O(\log^* N)$ by emulating each $\log^2 N \times \log^2 N$ mesh recursively. A more sophisticated scheme for real-time emulation is presented in [40].

3.5.3 Embedding the shuffle-exchange graph in the butterfly

In this section, we show how to embed an N-node shuffle-exchange graph in an $O(N)$-node butterfly graph with constant load, constant congestion, and $O(\log N)$ dilation. These graphs are defined in Sections 1.7 and 1.5, respectively.

A constant congestion embedding requires that very few edges of the shuffle-exchange graph be mapped to long (more than constant length) paths in the butterfly. In addition, these paths must not overlap each other very often. To ensure this, we use Waksman's observation [68] that the inputs and outputs of a Benes network can be connected in any permutation by a set of disjoint paths. That is, if the set of long paths can be decomposed into a constant number of (partial) permutations of the inputs of the butterfly, the long paths can be embedded with constant congestion. It is easy to see that we can embed the long paths in this manner when there are at most a constant number of endpoints of long paths in any single butterfly row. (We first route a path from each endpoint to the input of its row, which leaves us with a constant number of permutations to route on the Benes network.)

We map the nodes of a shuffle-exchange graph to the nodes of a butterfly graph so that

1. at most a constant number of shuffle-exchange nodes are mapped to any one butterfly node, and
2. each butterfly row contains at most a constant number of shuffle-exchange nodes which have any neighbor mapped to a distant node in the butterfly.

Short paths only contribute constant congestion since they have constant length. Long paths only contribute constant congestion since we can route any permutation with congestion 2, and we only need to route a constant number of (partial) permutations. Also, the length of the short paths is constant and the long paths is $O(\log n)$.

In particular, we map the nodes of a $N = 2^n$-node shuffle-exchange graph to the nodes of a $(n+2-\log n)2^{n+2-\log n} \approx 4N$-node butterfly graph. Each node in this N-node shuffle-exchange graph has n bits in its label. A node in the butterfly can be specified by a row represented by $n+2-\log n$ bits, and a level in the row. The level in the row corresponds to a bit that can be flipped to enter another row. Thus, we first associate a shuffle-exchange node with a particular row of the butterfly by removing $\log n - 1$ adjacent bits of its label none of which are the least

significant bit, then we pick the level in the row which corresponds to where the least significant bit of the shuffle-exchange node appears in the row's representation.

We map a shuffle-exchange node w to a node in the butterfly as follows,

1. Consider the longest string of zeros in w ignoring the least significant bit, break ties by choosing the first one from the left.

2. Pick out $\log n - 1$ bits as follows;

   (a) If possible choose the $\log n - 1$ bits after the zeros and before the lsb,

   (b) otherwise if possible choose the $\log n - 1$ bits preceding the longest string of zeros,

   (c) otherwise choose the last $\log n - 1$ bits of the string of zeros (note that in this case more than $n - 2\log n$ bits are zeros).

3. Treat these bits as a number (it will be in the range $0...\frac{n}{2}$), call this number s, and the sequence of bits a_s.

4. Remove the bits of s from w, extend the chosen string of zeros on the right (left) by a 01 (10) if the bits were removed from the right (left) of the block of zeros, and cyclic shift the resulting string so that s bits appear after the longest string of zeros, this specifies the row.

Symbolically, we map $w = z0^k a_s yb$ to row $u0^{k+1}1v$, or we map $w = za_s0^k yb$ to row $u10^{k+1}v$, with $ybz = vu$ and $|v| = s$. (Note that we map to a row with a unique longest string of zeros not straddling the bit which is at the level of the butterfly node.) It is easy to see that the least significant bit of w, b, is somewhere in the representation of the row. We choose the level in the row to correspond to the position of b in the row's representation.

We must argue that the mapping achieves condition 1 and 2 above.

First, we introduce some more notation. We define a *necklace* to be a set of shuffle-exchange nodes which are connected only by shuffle edges. Alternatively, a necklace is a set of nodes having labels which are cyclic shifts of each other. A necklace's *label* is the lexicographically minimum label of its nodes. We can specify a shuffle-exchange node by the label of its necklace and the position of the least significant bit of the node's label in the necklace's label.

We define the *domain* of a butterfly node to be the set of shuffle-exchange nodes that are mapped to it by our mapping.

Now we show that the mapping is at most two to one. That is, given a butterfly node $\langle p, r\rangle$ we can describe at most two shuffle-exchange nodes that could possibly be mapped to $\langle p, r\rangle$ as follows. Recall that a butterfly node $\langle p, r\rangle$ has all the bits of w in r's binary representation except for a_s. And these, we recover by finding the length of the string after the longest group of zeros in r's binary representation not straddling the pth bit. We know that we have to reinsert them either directly before or directly after that group of zeros. This gives us all the bits of the domain nodes except for a cyclic shift uncertainty. Thus, the domain of $\langle p, r\rangle$ can only be nodes from two necklaces. Furthermore, the least significant bit of the nodes' labels is uniquely specified by the place where the pth bit of r's binary representation occurs in the necklaces' labels. Thus only two shuffle-exchange nodes can be mapped to any node in the butterfly.

Finally, we argue that we map at most a constant number of shuffle exchange nodes with distant neighbors to any butterfly row.

Notice that we always ignore the value of the least significant bit in the mapping of shuffle-exchange nodes to butterfly nodes. Thus the mapping maps two shuffle-exchange nodes to two nodes that only differ in the bit that can currently be changed by a butterfly edge. Thus, any exchange edge needs only flip the bit at the node's level, which only requires a path of length 2. Thus all exchange edges are embedded in short paths.

Now consider the shuffle edges. We show that at most a constant number of shuffle edges leave any row of the butterfly. (It is easy to see that all the shuffle edges in a row are mapped to single edges in the butterfly graph.) Again, consider the inverse mapping of a butterfly node, $\langle p, r\rangle$, to two shuffle-exchange nodes. The necklaces of the domain nodes of row r's nodes, are the same for most of the row. They change only at certain transition levels in the row; levels, p, in the row where the position of the longest string of zeros not straddling p changes, or levels in the row where we become unsure or sure of which side of the zeros to replace the removed bits, a_s.

The position of the longest string of zeros not straddling p only changes at two points; inside the row's unique longest string of zeros. When the row level is within $\log n$ bit positions to the right of the longest string of zeros, we know that pieces of two shuffle-exchange necklaces could

have been mapped to the row. Outside this range we know that only one necklace is mapped to the row: Inside the group of zeros the bits were definitely taken out before the group of zeros, and further to the right they were definitely taken out after the group of zeros. Thus entering this stretch and leaving this stretch gives us two more bad levels. Thus we have four transition levels in all, and for each of these at most four necklaces could enter or leave the row at any of these levels. Thus at most 16 long shuffle edges can have endpoints in this row. (Careful counting can reduce this number to 6.)

Thus at most 16 long edges are adjacent to any row of the butterfly. This satisfies condition 2, above.

Thus, the shuffle-exchange graph can be embedded in the butterfly with constant congestion.

3.5.4 Layouts for the shuffle-exchange graph with optimal area and volume

The N-node butterfly can be laid out in $O(N^2/\log^2 N)$ area (trivially) and in $O(N^{3/2}/\log^{3/2} N)$ volume [100]. Since the N-node shuffle-exchange graph can be embedded in the N-node butterfly with constant congestion, we can simply blowup these layouts by a constant factor to obtain layouts for the shuffle-exchange graph with equivalent area and volume.

3.5.5 A work preserving emulation of a shuffle-exchange graph

We construct an $O(\log N)$- step work-preserving simulation of the shuffle-exchange graph on the butterfly by first embedding the shuffle-exchange graph in an $N \log N$-node butterfly with constant congestion, and then embedding the $N \log N$-node butterfly in an N-node butterfly in the natural way. It is not difficult to show that the N-node butterfly can then simulate the $N \lg N$-node shuffle-exchange in $O(\log N)$ steps. Whether or not there is a real-time emulation remains an interesting open question.

3.6 Emulations by shuffle-exchange graphs

3.6.1 Work preserving emulations of arbitrary binary trees

It is well known that the shuffle-exchange graph can emulate a complete binary tree in real time. Thus by the results of Section 3.4, we know that there is an $O(\log \log N)$-time work-

preserving emulation of the class of binary trees on the shuffle-exchange graph. Whether or not this emulation can be made real-time remains an open question.

3.6.2 Embedding little butterflies in the shuffle-exchange graph

In this section we show how to embed $M/\log M$ distinct $M \log M$-node butterfly graphs in an $N = M^2$ shuffle-exchange graph with load $l = 2$, congestion $c = O(1)$, and dilation $d = 3$. A similar result was proved by Raghunathan and Saran [80]. We assume that $M = 2^k$. Thus each row of the butterfly can be represented by a k-bit string, and each node of the shuffle-exchange can be represented by a $2k$-bit string.

To map $M/\log M$ butterflies to the shuffle-exchange graph, we use the following easily proven lemma.

Lemma 71 *The set of $k = \log M$-bit strings has at least $M/2\log M$ disjoint subsets each containing $\log M$ distinct strings which are cyclic shifts of each other.*

For each of these subsets we pick the lexicographically minimum string to represent the subsets. We associate the $M/\log M$ butterflies two to one with the $M/2\log M$ representative strings. Say butterfly i is associated with string w^i. We map a node $\langle p, r\rangle$ in butterfly i to a shuffle-exchange node by shuffling the bits of w_i with the bits of r's representation, and choosing the current bit to be under the image of r_p. That is, node $\langle p, r\rangle$ in butterfly i is mapped to shuffle-exchange node $r_1 w_1^i \ldots \underline{r_p} w_p^i \ldots r_n w_n^i$.

From a shuffle-exchange node we can recover the representative string w_i by picking out every other bit and shifting to the lexicographically minimum string. We find the row string by picking out the other bits and shifting by the same amount. The position in the row is clearly the number of shifts we used to get to w_i and the row number.

To finish, we observe that each edge in any of the butterflies is mapped to a path of length at most three in the shuffle-exchange graph since we either shift twice to reach $\langle p+1, r\rangle$'s image, or we exchange the current bit and shift twice to reach $\langle p+1, r_1 \ldots \overline{r_p} \ldots r_n\rangle$'s image.

Thus we can embed $\sqrt{N}/\log\sqrt{N}$ $(\sqrt{N}\log\sqrt{N})$-node butterflies in an N-node shuffle-exchange with load 2, congestion $O(1)$, and dilation 3.

This technique can be extended to prove that for any constant $0 < \epsilon < 1$, N^ϵ distinct $N^{1-\epsilon}$ butterfly graphs can be embedded in an N-node shuffle-exchange.

3.6.3 Application to sorting on a shuffle-exchange graph

It is known that an N-node butterfly can sort N packets with high probability in $O(\log N)$ steps [53, 76, 84]. The result does not directly extend to the shuffle-exchange graph because the shuffle-exchange graph does not have the nice recursive structure possessed by the butterfly. However, by combining the embedding result of Section 3.6.2, the butterfly sorting algorithm in [53], and the columnsort algorithm of [47], we can obtain an algorithm for sorting N packets on an N-node shuffle-exchange in $O(\log N)$ steps with high probability.

3.6.4 Real time emulations of arrays

By combining a single level of the kind of analysis in Section 3.5.3 with the result of Section 3.6.2, we can emulate an array in real time on a shuffle-exchange graph. This is despite the fact that any $O(1)$ to 1 embedding of an N-node array (with dimension 2 or more) in a shuffle exchange graph has dilation $\Omega(\log \log N)$ [11].

3.6.5 A work preserving emulation of the butterfly

By using standard techniques in routing normal hypercube algorithms, it is easily shown that there is an $O(\log N)$-step work-preserving simulation of a butterfly on a shuffle-exchange graph. Whether or not there is a real-time simulation remains an important open question.

Chapter 4

Minimum-cost spanning tree

4.1 Introduction

In this chapter we show that minimum-cost spanning tree is a special case of the closed semiring path-finding problem [1, sections 5.6–5.9]. For a graph of n vertices, the path-finding problem can be solved sequentially in $O(n^3)$ steps by a dynamic programming algorithm [37, 66] of which the algorithms of Floyd [25] and Warshall [99] are special cases. This dynamic programming algorithm has a well known $O(n)$ step implementation on an $n \times n$ mesh-connected computer [6, 19, 22, 30, 86].

Previously known minimum-cost spanning tree algorithms for the mesh [6, 61] are based on the recursive algorithm of Boruvka (also attributed to Sollin) [91, pp. 71–83], which is complicated to implement. For example, the algorithm of [6] achieves $O(n)$ steps by reducing the fraction of the mesh in use by a constant factor at each recursive call. The dynamic programming algorithm has the same asymptotic running time but is much simpler.

The rest of this chapter consists of two short sections. In Section 4.2 we show how to cast minimum-cost spanning tree as a path-finding problem. In Section 4.3, we briefly describe an $O(n)$ step mesh algorithm to solve the problem.

This chapter describes joint research with Serge Plotkin [65].

4.2 Reduction to a path-finding problem

In this section we define the minimum-cost spanning tree problem and a related path-finding problem. We give a recurrence for solving the path-finding problem via dynamic programming. We then prove that the solution to the path-finding problem contains the solution to the minimum-cost spanning tree problem.

Given an n-node connected [1] undirected graph $G = (V, E)$, where V is the set $\{1, \ldots n\}$, and where each edge $\{i, j\}$ in E has cost $C_{ij}^0 = C_{ji}^0$, the minimum-cost spanning tree problem is to find a subgraph that connects the vertices in V such that the sum of the costs of the edges in the subgraph is minimum. We assume that the edge costs are unique. (If not, lexicographical information can be added to make them unique.) For convenience, we also assume that if $\{i, j\}$ is not in E then it has cost $C_{ij}^0 = C_{ji}^0 = \infty$.

The path-finding problem is to compute the cost C_{ij}^k for each $1 \leq i, j, k \leq n$ of the shortest (lowest-cost) path from i to j that passes through vertices only in the set $\{1, \ldots, k\}$, where *the cost of a path is defined to be the highest cost of any edge on the path.* For any i and j, the shortest path from i to j with no intermediate vertex higher than k either passes through k or does not. In the first case, the cost of the shortest path from i to j is either the cost of the shortest path from i to k or the cost of the shortest path from k to j, whichever is higher. In the second case, we have $C_{ij}^k = C_{ij}^{k-1}$. Thus, C_{ij}^k can be computed by the recurrence

$$C_{ij}^k = \min\{C_{ij}^{k-1}, \max\{C_{ik}^{k-1}, C_{kj}^{k-1}\}\}.$$

The following theorem shows that the unique minimum-cost spanning tree can be recovered from the costs of the shortest paths.

Theorem 72 *An edge $\{i, j\}$ is in the unique minimum-cost spanning tree if and only if $C_{ij}^0 = C_{ij}^n$.*

Proof: The proof has two parts. We first show that if $\{i, j\}$ is a tree edge then $C_{ij}^0 = C_{ij}^n$. We then show that if $C_{ij}^0 = C_{ij}^n$ then the edge $\{i, j\}$ is in the tree. First, assume that $\{i, j\}$ is a tree edge, but that $C_{ij}^0 \neq C_{ij}^n$. Consider the cut of the graph that $\{i, j\}$ crosses, but no other

[1] For simplicity, we assume that the graph is connected. The same technique will find a minimum-cost spanning forest of a disconnected graph.

tree edge crosses. Since $C_{ij}^0 \neq C_{ij}^n$, there must be some path from i to j whose highest-cost edge has cost $C_{ij}^n < C_{ij}^0$. Hence, every edge on this path has cost less than C_{ij}^0. This path must cross the cut at least once. Replacing the edge $\{i,j\}$ by any edge on the path that crosses the cut reduces the cost of the tree, a contradiction. Conversely, assume that $C_{ij}^0 = C_{ij}^n$, but that $\{i,j\}$ is not a tree edge. Adding the edge $\{i,j\}$ to the tree forms a cycle whose highest-cost edge costs more than than C_{ij}^0. Replacing this edge by $\{i,j\}$ yields a tree with smaller cost, a contradiction. □

4.3 Implementation on a mesh-connected computer

In this section we give a short description of an $O(n)$ step algorithm for solving the minimum-cost spanning tree problem on an $n \times n$ mesh-connected computer. We assume that the diagonal element in each mesh row can broadcast a value to the other elements of the row in a single step. This type of broadcast can be simulated by a mesh without this capability by slowing the algorithm down by a constant factor [45, 59, 60]. The algorithm proceeds as follows. We assume that the input graph is given in the form of a matrix of edge costs C^0 which enters row-by-row through the top of the mesh. Matrix row i is modified as it passes over rows 1 through $i-1$ and is stored when it reaches mesh row i. When matrix row i passes over mesh row k, the value C_{ik}^{k-1} is broadcast right and left from the diagonal cell (k,k). Each cell $(k,j), 1 \leq j \leq n$ knows the value of C_{kj}^{k-1} and computes

$$C_{ij}^k = \min\{C_{ij}^{k-1}, \max\{C_{ik}^{k-1}, C_{kj}^{k-1}\}\}.$$

which is passed down to the next mesh row. After reaching mesh row i, matrix row i stays there until each matrix row l, $i < l \leq n$, above it has passed over it and then continues to propagate down, passing over the rest of the matrix rows. The output matrix C^n exits row-by-row from the bottom of the mesh. By Theorem 72, the adjacency matrix of the minimum-cost spanning tree can be constructed by comparing the input and output matrices.

Directions for further research

Packet routing algorithms, distributed random-access machines, and network emulations are the objects of ongoing research. This section presents some of the open questions and very recent results in these areas.

Packet routing

Many challenging routing and sorting problems remain to be solved. As we mentioned in Section 1.2, there is no efficient algorithm known for finding a schedule of length $O(c+d)$ for a set of packets whose paths have congestion c and dilation d. Also, there is no known algorithm simpler than that of Section 1.5 for routing on an N-node butterfly in $O(\log N)$ steps using constant-size queues. A simple FIFO queueing discipline performs well in simulations but has eluded analysis.

Although Sections 1.9 and 3.6 provide randomized algorithms for sorting on the butterfly and shuffle-exchange graphs in $O(\log N)$ steps using constant-size queues, there are no known deterministic algorithms for routing or sorting on the butterfly or shuffle-exchange graphs in $O(\log N)$ steps, even if large queues are allowed. Recently Cyper and Plaxton [21] discovered a deterministic algorithm for sorting on the shuffle-exchange graph in $O(\log N(\log\log N)^2)$ steps. Also, Upfal [95] recently found a deterministic algorithm for routing on a *multibutterfly* network in $O(\log N)$ steps using constant-size queues. However, Upfal's algorithm does not combine multiple packets with the same destination. The only known deterministic algorithm for sorting N packets on an N-node bounded-degree network in $O(\log N)$ steps [47] is based on the complicated AKS network [2].

Routing in the presence of faults has become an area of intense research. Typically it

is assumed that some of the edges or some of the nodes cannot transmit packets, and that these failures are easily detected. It is also sometimes assumed that the faults are distributed randomly throughout the network. Some of the recent results are summarized below.

In 1987, Hastad, Leighton, and Newman [31] presented a simple randomized on-line algorithm for embedding an N-node hypercube in N-node faulty hypercube with constant load, congestion, dilation. Faults are assumed to occur at the nodes randomly and independently with some fixed probability p. As a consequence, the faulty hypercube can emulate a fault-free hypercube with constant slowdown. Thus, it can route any permutation of N packets in $O(\log N)$ time using constant-size queues on the edges. In 1989 they discovered an $O(\log N)$-step algorithm for routing directly on the faulty hypercube [32]. The algorithm is *adaptive* in the sense that packets alter their paths to avoid faults.

Rabin [77] designed a fault-tolerant routing algorithm for the hypercube using error-correcting codes. His idea is to break each message into smaller pieces and encode them so that the original message can be recovered from any majority of them. In the course of routing, pieces are lost if they attempt to use faulty edges, enter full queues, or fail to reach their destinations quickly. Despite these losses, with high probability a majority of the pieces for each message reach their destinations. The algorithm routes a permutation of N messages in $O(\log N)$ time on an N-node hypercube using constant-size queues at the nodes. Edges are assumed to fail randomly and independently with probability $1/\log^2 N$. In this scheme, each message is broken into $\log N$ pieces. Attached to each piece is a $\Theta(\log N)$-bit ticket of error-correcting information. Thus, for the scheme to be efficient, messages must be at least $\Omega(\log^2 N)$ bits long.

Raghavan [79] considered routing permutations on a faulty mesh. He showed that on a $\sqrt{N} \times \sqrt{N}$ mesh where nodes fail randomly and independently with some fixed probability $p \leq .29$, every packet that can reach its destination does so in $O(\sqrt{N} \log N)$ time. The algorithm is randomized and uses queues of size $O(\log^2 N)$. Raghavan's result was improved by Karlin, Leighton, Raghavan, and Thomborson, who showed that after sustaining k faults, a mesh can route any permutation in $\min\{\sqrt{N} + O(k^2), N\}$ time.

In [52] we described an adaptive algorithm for routing on Upfal's multibutterfly [95] in the presence of faults. We proved that an N-input multibutterfly can sustain k faults and still route

$\log N$ permutations between some set of $N - O(k)$ inputs and $N - O(k)$ outputs in $O(\log N)$ time. The multibutterfly is even more resilient to randomized faults. A specially modified twin butterfly can tolerate $N^{3/4}$ faults at internal nodes, and still route any $\log N$ permutations of N packets in $O(\log N)$ time. Before routing begins, faulty regions are spliced out of the multibutterfly. Thereafter, the packets route as if there were no faults.

Distributed random-access machines

To date, all DRAM algorithms solve graph theoretic problems. It is natural to wonder whether there are other problem domains for which communication-efficient algorithms can be designed. One difficulty faced in designing DRAM algorithms for other domains is the lack of uniform-cost shared memory in the model. Unfortunately we haven't found any meaningful way to incorporate PRAM-like memory into the model.

Emulations

Chapter 3 leaves open several challenging problems. For example, we do not know if there a real-time simulation of a complete ternary tree on a complete binary tree. Another unresolved question is whether there is a class of bounded-degree graphs that can efficiently emulate the class of all bounded-degree graphs. If so, the graphs in this universal class must be expanders.

Schwabe recently resolved a long open question by proving that the butterfly and shuffle-exchange graphs are computationally equivalent[85]. He showed that each network can perform a real-time emulation of the other. The proof combines the techniques of embedding little butterflies in a shuffle-exchange graph from Section 3.6 (and vice versa) with the overlap strategy from Section 3.5.2 used by the butterfly to emulate the mesh.

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[2] M. Ajtai, J. Komlos, and E. Szemeredi. An $O(N \log N)$ sorting network. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, pages 1–9, April 1983.

[3] R. Aleliunas. Randomized parallel communication. In *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 60–72, August 1982.

[4] D. Angluin and L. G. Valiant. Fast probabilistic algorithms for hamiltonian circuits and matchings. *Journal of Computer and System Sciences*, 18(2):155–193, April 1979.

[5] M. Atallah and U. Vishkin. Finding Euler tours in parallel. *Journal of Computer and System Sciences*, 29(3):330–337, July 1984.

[6] M. J. Atallah and S. R. Kosaraju. Graph problems on a mesh-connected processor array. *Journal of the ACM*, 31(3):649–667, July 1984.

[7] B. Awerbuch, A. Israeli, and Y. Shiloach. Finding Euler circuits in logarithmic parallel time. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, pages 249–257, April 1984.

[8] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for Ultracomputer and PRAM. In *Proceedings of the 1983 International Conference on Parallel Processing*, pages 175–179. IEEE, August 1983.

[9] K. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computing Conference*, volume 32, pages 307–314, 1968.

[10] V. E. Benes. *Mathematical Theory of Connecting Networks and Telephone Traffic*. Academic Press, New York, 1965.

[11] S. N. Bhatt, F. R. K. Chung, J.-W. Hong, F. T. Leighton, and A. L. Rosenberg. Optimal simulations by butterfly networks. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pages 192–204, May 1988.

[12] S. N. Bhatt, F. R. K. Chung, F. T. Leighton, and A. L. Rosenberg. Optimal simulations of tree machines. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, pages 274–282. IEEE, October 1986.

[13] S. N. Bhatt and I. Ipsen. Embedding trees in the hypercube. Technical Report RR-443, Yale University, New Haven, CT, 1988.

[14] S. N. Bhatt and C. E. Leiserson. How to assemble tree machines. In F. P. Preparata, editor, *VLSI Theory*. Volume 2 of *Advances in Computing Research*, pages 95–114. JAI Press, Greenwich, CT, 1984.

[15] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–208, April 1974.

[16] R. P. Brent and H. T. Kung. A regular layout for parallel adders. *IEEE Transactions on Computers*, C-31(3):260–264, March 1982.

[17] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, April 1979.

[18] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, 23:493–507, 1952.

[19] T. W. Christopher. An implementation of Warshall's algorithm for transitive closure on a cellular computer. Technical Report 36, Institute for Computer Research, University of Chicago, Chicago, IL, 1973.

[20] R. Cole and U. Vishkin. Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pages 206–219, May 1986.

[21] R. E. Cypher and C. G. Plaxton. Deterministic sorting in nearly logarithmic time on the hypercube and related computers. Unpublished manuscript.

[22] E. Dekel, D. Nassimi, and S. Sahni. Parallel matrix and graph algorithms. *SIAM Journal on Computing*, 10(4):657–675, November 1981.

[23] A. K. Dewdney. Computer recreations. *Scientific American*, 252(6):18–29, June 1985.

[24] M. J. Fischer and R. E. Ladner. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, October 1980.

[25] R. W. Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[26] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pages 136–146, May 1986.

[27] A. V. Goldberg and R. E. Tarjan. Solving minimum-cost flow problems by successive approximation. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 7–18, May 1987.

[28] D. S. Greenberg, L. S. Heath, and A. L. Rosenberg. Optimal embeddings of the FFT graph in the hypercube. Unpublished manuscript.

[29] R. I. Greenberg and C. E. Leiserson. Randomized routing on fat-trees. In Silvio Micali, editor, *Randomness and Computation*. Volume 5 of *Advances in Computing Research*. JAI Press, Greenwich, CT, 1989. To appear.

[30] L. J. Guibas, H. T. Kung, and C. D. Thompson. Direct VLSI implementation for combinatorial algorithms. In C. L. Seitz, editor, *Proceedings of the Caltech Conference on Very Large Scale Integration*, pages 509–525, Pasadena, CA, January 1979. Caltech Computer Science Department.

[31] J. Hastad, T. Leighton, and M. Newman. Reconfiguring a hypercube in the presence of faults. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 274–284, May 1987.

[32] J. Hastad, T. Leighton, and M. Newman. Fast computation using faulty hypercubes. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 251–263, May 1989.

[33] W. D. Hillis and G. L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[34] D. Hoey and C. E. Leiserson. A layout for the shuffle-exchange network. In *Proceedings of the 1980 International Conference on Parallel Processing*, pages 329–336. IEEE, August 1980.

[35] A. R. Karlin and E. Upfal. Parallel hashing — an efficient implementation of shared memory. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pages 160–168, May 1986.

[36] J. Kilian, July 1986. Private communication.

[37] S. C. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–41. Princeton University Press, Princeton, NJ, 1956.

[38] D. J. Kleitman, F. T. Leighton, M. Lepley, and G. L. Miller. New layouts for the shuffle-exchange graph. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, pages 278–292, May 1981.

[39] D. E. Knuth. *The Art of Computer Programming*, volume 1. Addison–Wesley, Reading, MA, second edition, 1973.

[40] R. Koch, T. Leighton, B. Maggs, S. Rao, and A. Rosenberg. Work-preserving emulations of fixed-connection networks. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 227–240, May 1989.

[41] D. Krizanc, S. Rajasekaran, and Th. Tsantilis. Optimal routing algorithms for mesh-connected processor arrays. In J. Reif, editor, *Aegean Workshop on Computing: VLSI Algorithms and Architectures.* Volume 319 of *Lecture Notes in Computer Science*, pages 411–422. Springer–Verlag, New York, NY, June 1988.

[42] C. P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. Unpublished manuscript.

[43] M. Kunde. Routing and sorting on mesh-connected arrays. In J. Reif, editor, *Aegean Workshop on Computing: VLSI Algorithms and Architectures*. Volume 319 of *Lecture Notes in Computer Science*, pages 423–433. Springer–Verlag, New York, NY, June 1988.

[44] H. T. Kung and C. E. Leiserson. Systolic arrays (for VLSI). In I. S. Duff and G. W. Stewart, editors, *Sparse Matrix Proceedings*, pages 256–282. SIAM, 1978.

[45] F. T. Leighton. An introduction to the theory of networks, parallel computation and VLSI design. Unpublished manuscript.

[46] F. T. Leighton. *Complexity Issues in VLSI*. MIT Press, Cambridge, MA, 1983.

[47] F. T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, C-34(4):344–354, April 1985.

[48] F. T. Leighton, M. Lepley, and G. L. Miller. Layouts for the shuffle-exchange graph based on the complex plane diagram. *SIAM Journal of Algebraic and Discrete Methods*, 5:177–181.

[49] F. T. Leighton, F. Makedon, and I. Tollis. A $2N - 2$ step algorithm for routing in an $N \times N$ mesh. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 328–335, June 1989.

[50] F. T. Leighton and G. L. Miller. Optimal layouts for small shuffle-exchange graphs. In J. Gray, editor, *VLSI 81-Very Large Scale Integration*, pages 289–299. Academic Press, 1981.

[51] F. T. Leighton and A. L. Rosenberg. Three-dimensional circuit layouts. *SIAM Journal on Computing*, 15(3):793–813, August 1986.

[52] T. Leighton and B. Maggs. Expanders might be practical: fast algorithms for routing around faults in multibutterflies. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 384–389. IEEE, October 1989.

[53] T. Leighton, B. Maggs, and S. Rao. Universal packet routing algorithms. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 256–271. IEEE, October 1988.

[54] T. Leighton and S. Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 422–431. IEEE, October 1988.

[55] C. E. Leiserson. *Area-Efficient VLSI Computation*. MIT Press, Cambridge, MA, 1983.

[56] C. E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C-34(10):892–901, October 1985.

[57] C. E. Leiserson and B. M. Maggs. Communication-efficient parallel graph algorithms. Technical Memo MIT/LCS/TM-318, MIT Laboratory for Computer Science, Cambridge, MA, December 1986.

[58] C. E. Leiserson and B. M. Maggs. Communication-efficient parallel graph algorithms for distributed random-access machines. *Algorithmica*, 3:53–77, 1988.

[59] C. E. Leiserson, F. M. Rose, and J. B. Saxe. Optimization of synchronous circuitry by retiming. In R. Bryant, editor, *Third Caltech Conference on Very Large Scale Integration*, pages 87–116, Rockville, MD, March 1983. Computer Science Press.

[60] C. E. Leiserson and J. B. Saxe. Optimizing synchronous systems. *Journal of VLSI and Computer Systems*, 1(1):41–46, 1983.

[61] K. N. Levitt and W. H. Kautz. Cellular arrays for the solution of graph problems. *Communications of the ACM*, 15(9):789–801, September 1972.

[62] R. J. Lipton and R. E. Tarjan. A planar separator theorem. *SIAM Journal of Applied Mathematics*, 36(2):177–189, April 1979.

[63] B. M. Maggs. A scheme for area-universal computation. Unpublished manuscript.

[64] B. M. Maggs. Communication-efficient parallel graph algorithms. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 1986.

[65] B. M. Maggs and S. A. Plotkin. Minimum-cost spanning tree as a path-finding problem. *Information Processing Letters*, 26(6):291–293, January 1988.

[66] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IRE Transactions on Electronic Computers*, 9(1):39–47, 1960.

[67] F. Meyer auf der Heide. Efficient simulations among several models of parallel computers. *SIAM Journal on Computing*, 15(1):106–119, February 1986.

[68] G. Miller and J. Reif. Parallel tree contraction and its application. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, pages 478–489. IEEE, October 1985.

[69] R. Miller, V. K. Prasanna-Kumar, D. Reisis, and Q. F. Stout. Meshes with reconfigurable buses. In J. Allen and F. T. Leighton, editors, *Advanced Research in VLSI: Proceedings of the Fifth MIT Conference*, pages 163–178, Cambridge, MA, April 1988. MIT Press.

[70] D. Nassimi and S. Sahni. Parallel permutation and sorting algorithms and a new generalized connection network. *Journal of the ACM*, 29(3):642–667, July 1982.

[71] Yu. Ofman. On the algorithmic complexity of discrete functions. *Soviet Physics – Doklady*, 7(7):589–591, 1963. English translation.

[72] C. H. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pages 510–513, May 1988.

[73] J. K. Park. A deterministic routing algorithm for the butterfly fat-tree. Unpublished manuscript.

[74] D. Peleg and E. Upfal. The token distribution problem. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, pages 418–427. IEEE, October 1986.

[75] G. F. Pfister and V. A. Norton. 'Hot spot' contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, C-34(10):943-948, October 1985.

[76] N. Pippenger. Parallel communication with limited buffers. In *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*, pages 127–136. IEEE, October 1984.

[77] M. O. Rabin. Efficient dispersal of information for security load balancing and fault tolerance. *Journal of the ACM*, 1989. To appear.

[78] P. Raghavan. Probabilistic construction of deterministic algorithms: approximate packing integer programs. *Journal of Computer and System Sciences*, 37(4):130–143, October 1988.

[79] P. Raghavan. Robust algorithms for packet routing in a mesh. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 344–350, June 1989.

[80] A. Raghunathan and H. Saran. Is the shuffle-exchange better than the butterfly? Unpublished manuscript.

[81] A. G. Ranade. How to emulate shared memory. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, pages 185–194. IEEE, October 1987.

[82] A. G. Ranade. *Fluent Parallel Computation.* PhD thesis, Yale University, New Haven, CT, 1988.

[83] J. H. Reif. Personal communication.

[84] J. H. Reif and L. G. Valiant. A logarithmic time sort for linear size networks. *Journal of the ACM*, 34(1):60–76, January 1987.

[85] E. J. Schwabe. The butterfly and shuffle-exchange graph are computationally equivalent. Unpublished manuscript.

[86] F. L. Van Scoy. The parallel recognition of classes of graphs. *IEEE Transactions on Computers*, C–29(7):563–570, July 1980.

[87] M. Sekanina. On an ordering of the set of vertices of a connected graph. *Publications of the Faculty of Science, University of Brno*, 412:137–142, 1960.

[88] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3:57–67, 1982.

[89] J. Spencer. *Ten Lectures on the Probabilistic Method.* SIAM, Philadelphia, PA, 1987.

[90] D. Steinberg and M. Rodeh. A layout for the shuffle-exchange network with $\Theta(N^2/\log^{3/2} N)$ area. *IEEE Transactions on Computers*, C-30(12):977–982, December 1981.

[91] R. E. Tarjan. *Data Structures and Network Algorithms.* SIAM, Philadelphia, PA, 1983.

[92] R. E. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. In *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*, pages 12–20. IEEE, October 1984.

[93] C. D. Thompson. *A Complexity Theory for VLSI.* PhD thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1980.

[94] E. Upfal. Efficient schemes for parallel communication. In *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 55–59, August 1982.

[95] E. Upfal. An $O(\log N)$ deterministic packet routing scheme. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 241–250, May 1989.

[96] L. G. Valiant. A scheme for fast parallel communication. *SIAM Journal on Computing*, 11(2):350–361, May 1982.

[97] L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, pages 263–277, May 1981.

[98] A. Waksman. A permutation network. *Journal of the ACM*, 15(1):159–163, January 1968.

[99] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, January 1962.

[100] D. S. Wise. Compact layouts of Banyan/FFT networks. In H. T. Kung, B. Sproull, and G. Steele, editors, *CMU Conference on VLSI Systems and Computations*, pages 186–195, Rockville, MD, October 1981. Computer Science Press.

[101] J. C. Wyllie. *The Complexity of Parallel Computations.* PhD thesis, Cornell University, Ithaca, NY, August 1979.

OFFICIAL DISTRIBUTION LIST

Director Information Processing Techniques Office Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209	2 copies
Office of Naval Research 800 North Quincy Street Arlington, VA 22217 Attn: Dr. Gary Koop, Code 433	2 copies
Director, Code 2627 Naval Research Laboratory Washington, DC 20375	6 copies
Defense Technical Information Center Cameron Station Alexandria, VA 22314	12 copies
National Science Foundation Office of Computing Activities 1800 G. Street, N.W. Washington, DC 20550 Attn: Program Director	2 copies
Dr. E.B. Royce, Code 38 Head, Research Department Naval Weapons Center China Lake, CA 93555	1 copy